UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA
Departamento de Arquitectura de Computadores y Automática

A SET OF TECHNIQUES TO REDUCE PERFORMANCE AND ENERGY OVERHEADS FOR DYNAMIC RECONFIGURATION.

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Elena Pérez Ramo

Bajo la dirección de los doctores

Jesús Javier Resano Ezcaray
Daniel Mozos Muñoz

Madrid, 2010

- **ISBN: 978-84-693-3362-4**

Técnicas para reducir la penalización en rendimiento y consumo debido a la reconfiguración dinámica

A set of techniques to reduce performance and energy overheads for dynamic reconfiguration

---

Elena Pérez Ramo

Dissertation
Presented to the Facultad de Informática of the
Universidad Complutense de Madrid
In Partial Fulfilment
Of the Requirements
For the Degree of

Doctor of Philosophy

Universidad Complutense de Madrid

Dpto. Arquitectura de Computadores y Automática

2009

Técnicas para reducir la penalización en rendimiento y consumo debido a la reconfiguración dinámica

Memoria presentada por Elena Pérez Ramo para optar al grado de doctor con mención Europea por la Universidad Complutense de Madrid, realizada bajo la dirección de D. Jesús Javier Resano Ezcaray y D. Daniel Mozos Muñoz.

A set of techniques to reduce performance and energy overheads for dynamic reconfiguration

Dissertation presented by Elena Pérez Ramo in partial fulfilment of the requirements for the degree of Doctor of Philosophy with European Mention to the Universidad Complutense de Madrid, supervised by Mr. Jesús Javier Resano Ezcaray and Mr. Daniel Mozos Muñoz.

Madrid, Julio 2009.

OUTLINE

Chapter 1:
Importance of Embedded Computing

Embedded computing is becoming critical to many products that are mainstays of modern society. For example, automotive costs are now approaching the proportions long held by aircraft: the cost of an automobile is now about one-third frame, one-third power-plant, and one-third electronics. Automobiles could not meet the fuel efficiency and pollution targets set by law without sophisticated embedded controllers. First-generation cell phones used analogical transmission with simple microcontrollers to handle basic tasks, but modern cell phones rely on microprocessors for core tasks like voice compression and baseband signal processing, not to mention additional features like phone books and multimedia. Hence, the main difficulty is not just achieving the application requested performance

but achieving that performance in the face of three major requirements, namely, real time, low energy, and acceptable area cost:

- Real-time computing involves dealing with deadlines. Traditional desktop applications like word processing do not have to meet deadlines and they can just adopt a best effort approach. Hence, they can slow down when they perform some complex tasks or if other jobs are running on the same machine. However, in embedded systems a best effort approach is often not acceptable. For example, applications for automobile and digital still cameras, must deal with deadlines. In the case of a camera, a photograph may be ruined if the embedded system does not meet its deadlines; in the case of cars, a catastrophic accident could be the result of a missed deadline. These are two examples of hard deadlines. In these cases the system must guarantee that all these deadlines are met. Multimedia applications, like digital video applications and online games with 3D graphics, also include deadlines, since audio and video tasks must be synchronized and images must be visualized at a certain rate to achieve a proper quality level. In this case, these are softer deadlines since the system loses only some information if it does not meet one of these deadlines, so it will only slightly decrease its quality-of-service level. Hence, if might be acceptable that the system miss one every thousand deadlines.
- As we have mentioned before, in the case of portable battery-operated devices (PDAs, cell phones…), the system must not only deal with high

performance applications requirements, but also with demanding low-power constraints. For example cell phones must operate many hours while using batteries. Often these devices must deliver almost desktop-level performance at two or three orders of magnitude less power consumption.

Desktop computers can improve their performance or extend their functionality including new elements in the system like a graphic co-processor or HW support for a wireless communication standard. However, in embedded systems the area is very constrained and normally it is not possible to include new elements in the system.

1.1. New Multimedia Application Features

Nowadays a new generation of portable embedded systems (e.g. PDAs, mobile phones…) must deal with very exigent applications that often present a highly dynamic behaviour with a workload that varies several orders of magnitude at run-time. Hence, at a certain point of time they may demand high performance levels (for instance when a 3D rendering must deal with new complex 3D objects), while some time afterwards the workload may be drastically reduced (for instance for still images). These variations depend on several external factors, like the interaction among the application, the users, other applications and the input data. Hence, they are not predictable at design-time.

Typically, these applications were initially developed for general-purpose computers, which will provide a high performance, or for application-specific devices

specially designed for them. But nowadays these kinds of applications are being more and more demanded in embedded systems where available resources are highly limited due to their small size, their price, and their battery life.

This chapter explains the main features of the recent multimedia standards using the MPEG-4 standard ([KnSM99], [M4if05]) as an illustrative example.

Like all new standards, MPEG-4 involves new challenges for designers since they must deal with a wide number of different algorithms which demand high performance and at the same time present much more dynamic and unpredictable behaviour than the previous standards, like MPEG or MPEG-2.

While MPEG-2 was created as a specific standard for video and digital TV, MPEG-4 has been developed for a wider scope. It includes video support for small-bandwidth systems (covered before by the H.263 standard, for wireless technology), for medium-bandwidth systems (covered before by the MPEG-1 standard, designed for internet video), and finally for high-bandwidth systems (previously covered by the MPEG-2 standard, for digital video/TV). To sum up, the MPEG-4 supports applications that reproduce video between 64Kb/s and 10Mb/s.

Besides, MPEG-4 is not only a video standard. Video is only one of the multiple functionalities that an MPEG-4 application can include. MPEG-4 provides a standard for multimedia applications that use mobile phone or wireless communications, network and Internet communications, TV and radio broadcasts, image, video and sound codifications for a wide range of qualities, 2D and 3D graphics support

(including specific tools for human face and body animation), and finally the interactive integration of all these possibilities.

In contrast with previous standards designed for specific applications, MPEG-4 provides a framework to easily develop heterogeneous applications that include functionalities from different fields and that can interact with their environment. For example, a newspaper decides to develop a news server. The newspaper server has a home page that offers links to different contents. Besides, this home page can be accessed from a wide range of devices, like mobile phones, PDAs, personal computers through phone or high-bandwidth connections, or even through digital TV channels. On a traditional approach, different versions of this page should be developed, for each possible connection. Nevertheless, MPEG-4 allows using the same version of the home page for all of them. Thus, the different items that compose this page (videos, images, 3D graphics, text documents, etc…) are stored applying scalable coding algorithms. This allows sending the accurate information level for the specific receptor device. This device will receive all the items, with the precise quality, coupled with a scene descriptor that contains the information needed to correctly assemble and display the information. MPEG-4 describes an image as a composition of a non-fixed number of heterogeneous objects. Each one of these objects can be coded and encoded independently.

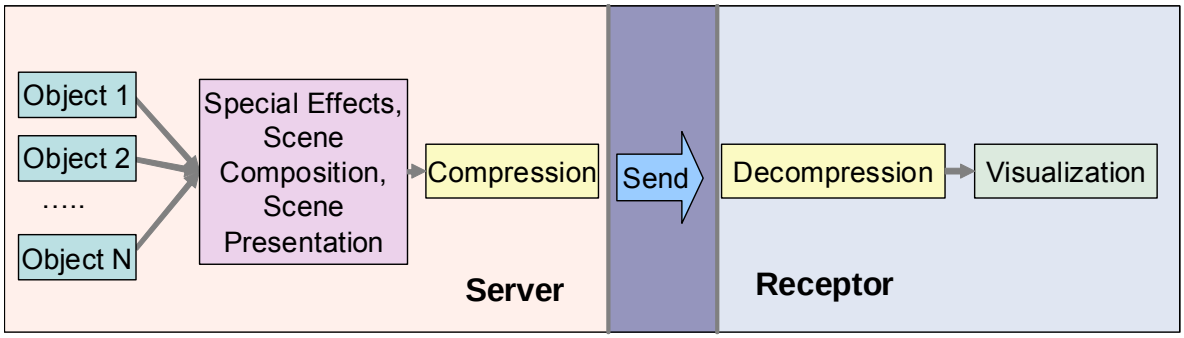


Figure 1. Conventional video codification scheme.

Figure 1 depicts how traditional video standards work. These standards work with only one image that covers the entire screen. If the image includes different items, they are previously combined to build up the final image.

As it is depicted in Figure 2, MPEG-4 works in a different way. Using MPEG-4, each object is individually stored, optimized and encoded. Next, the compressed objects are sent with the scene descriptor in BIFS format (Binary Format for Scenes), which is an MPEG-4 internal format. With all this information, and taking into account the possible interaction with the user, the receptor builds up the final scene.

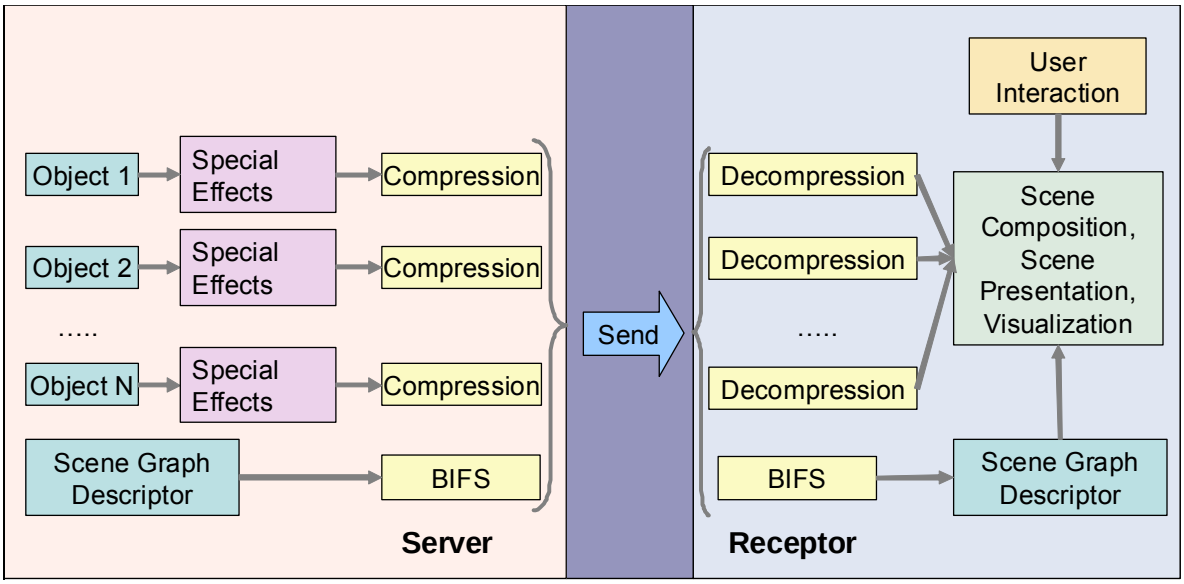


Figure 2. Based on objects codification scheme.

Working with objects enables the development of hybrid systems where quite heterogeneous items can be included. Nowadays video applications that include computer-generated figures are more and more common. On a classical video approach these figures must be previously converted to natural images, and then they have to be compressed using natural images compression algorithms. However, MPEG-4 achieves better results dealing with each component separately. Besides, if the application is a game, where the different figures are moved following the orders of a user, a classical video standard cannot manage that dynamic behaviour. However, MPEG-4 has no problems with that dynamism since it allows user interaction using the scene descriptor.

This object-based approach simplifies the extraction of enough parallelism to efficiently work with multiprocessor platforms, since the system is explicitly generating

a set of tasks with a high computational load and without data dependencies between them.

Another important consequence is that the execution workload of a MPEG-4 application may rapidly vary, even from one scene to another, since the number of objects and their type can change. Besides, even if the number and type of objects do not change, the computational load can do it due to user interaction. For instance the user may modify the visualization quality. MPEG-4 supports this option allowing that a decoder can use only a percentage of the received data. Hence, by increasing or decreasing this percentage the system can adapt the visualization quality. Usually the QoS level changes them.

Another consequence of this very heterogeneous standard is that the number of algorithms that an application must execute may be rather high, since it may support a wide number of different objects and qualities, and each combination may need different encoding/decoding algorithms. Many of these algorithms may demand high performance. However embedded processors are normally optimized for low power and probably could not meet the performance requirements. Hence, they will need some hardware accelerators to meet the deadlines and the required quality of service. Nevertheless, due to the tight area constraints, probably most embedded systems cannot afford to include ASICs to provide HW support for each one of these algorithms and it is necessary to find a more flexible solution that is still energy-efficient and meets the real-time constraints. In this context, reconfigurable HW resources or programmable Application Specific Instruction Processors (ASIPs) appear to be the most promising solutions available in the market [PTCC09] and

[PTMM09]. An ASIP is used in System-on-a-Chip design. The instruction set of an ASIP is tailored to benefit a specific application. This specialization of the core provides a trade-off between the flexibility of a general purpose CPU and the performance of an ASIC. Some ASIPs have a configurable instruction set. Usually, these cores are divided into two parts: static logic which defines a minimum ISA and configurable logic which can be used to design new instructions. The configurable logic is usually implemented in a FPGA.

1.2. Our Target Architecture

A flexible and powerful platform should take advantage of the available processing elements. Traditionally an embedded platform was built using two different types of resources: embedded processors and application-specific integrated circuits (ASICs).

Instruction set processors (ISPs) compute an application executing a sequence of instructions stored in a local memory. The most important advantage of this model is that it provides a high degree of flexibility since the processor can execute any application as long as the code is properly compiled. Nevertheless, this flexibility has some important drawbacks. First of all, the instructions to execute must be selected from a fixed set of instructions defined at design time. Therefore, if an operation is not available in this set it has to be implemented using a combination of the existing ones. In addition, all instructions must be decoded before execution, which also involves a significant overhead. Finally, embedded processors normally cannot take

full advantage of the inner parallelism of an algorithm. Due to these reasons, traditional embedded ISPs do not always provide the requested performance, especially since they are normally optimized for low power execution.

From the performance and energy consumption point of view, the best option to implement a specific algorithm is to include in the system an ASIC specifically designed to execute it. These ASICs will include HW optimized for the execution of the algorithm, in addition they will not need to carry out a costly decoding process, and they will size the HW resources to take advantage of the algorithm parallelism.

However, the limited area, and the increasing number of supported applications entails that a platform developer cannot afford ASICs as a general solution. For example, mobile phones have begun to include video, audio, and image coders and decoders, 3D games applications and different standards for wireless connections. On a personal computer these applications are optimized adding specific HW for them. But, in a embedded system there is not enough space to include all this HW support. Besides, it is more and more common that end users can increase the functionality of their systems, adding new applications not known at design time. Nevertheless, due to embedded system reduced size, it is not possible to add new HW that implements the new functionality. Moreover, developing a new ASIC is an expensive, time-consuming and error-prone task. Hence, although ASICs are very interesting to accelerate some critical algorithms, they are not a generic solution to build flexible embedded platforms.

Reconfigurable HW offers an interesting possibility to cover the existing gap between ASICs and general purpose processors, trying to gather the advantages of both platforms, while avoiding some of their limitations.

Figure 3 presents our target system: a heterogeneous multiprocessor SOC that includes several reconfigurable units. Firstly, it contains one or several ISPs that will provide some OS support and will execute most control computations and those tasks that do not demand high performance. In addition, the system will include some ASICs and especially more flexible ASIPs to execute the most critical computations known at design-time. Finally, it will include a set of reconfigurable units composed of some reconfigurable HW resources wrapped with a fixed interface that provides communication and synchronization support for the HW tasks implemented in the reconfigurable resources.

Apart from the processing elements the system will include communication resources like a system bus or a bus hierarchy, or an interconnection network that typically is a Network-on-Chip (NoC), and some specific circuitry to carry out the run-time reconfigurations.

Similar architectures have already succeeded in the multimedia market. The best example may be the Sony PSP™ architecture that includes two embedded processors R4000 processors (one of them with a vector processing unit), a Graphics Processing Unit (GPU), and a VME (Virtual Mobile Engine™ [Sony05]) that is a reconfigurable processor developed by SONY. SONY has included this reconfigurable processor in several portable platforms; since they have tested that it not only provides high performance but also consumes five times less power than

their embedded processors. The VME is not based on a FPGA-like architecture, it is basically a reconfigurable data-path sharing with our approach the idea of using reconfigurable HW as HW accelerator in a heterogeneous multiprocessor system.

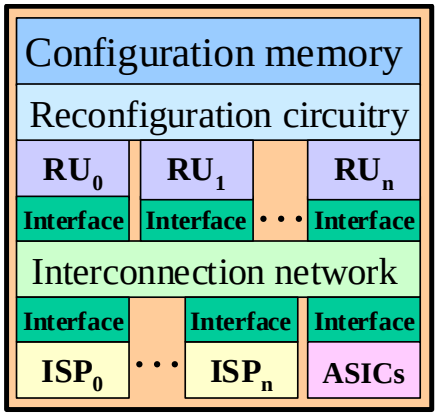


Figure 3. Example of target architecture.

1.3. Dynamic Reconfigurable HW

Reconfigurable resources can adapt their functionality at run-time to meet system demands by loading the proper configuration file. Each configuration file contains the information needed to implement a custom HW design using the reconfigurable HW. Hence the same set of reconfigurable resources can be used as a HW accelerator for different algorithms at different points of time.

Reconfigurable HW introduces the concept of virtual hardware, which is similar to the idea of virtual memory, allowing to swapping configurations in and out at run-time. Hence the physical hardware area can be much smaller than the addition of the HW area required by each configuration. This feature yields important area savings.

In addition, the designer does not need to develop all the configurations at design-time, since it is always possible to include new configurations in the system. This is very useful, for instance, in order to deal with new applications, or to include upgrades that optimize the system, or to correct some errors that have been detected.

Reconfigurable HW also introduces performance improvements over general-purpose processors, since it allows achieving the needed performance, taking advantages of the inherent parallelism of the application.

Hence, reconfigurable HW or ASIPs platforms have gathered most of the advantages of general purpose processors (flexibility, wide range of application execution, and cost), and at the same time provides a solution that can achieve performance levels and energy-efficiency almost as good as ASICs.

The runtime flexibility of reconfigurable HW often comes at the price of important reconfiguration penalties each time that a new task is assigned to a reconfigurable unit (RU). These reconfigurations involve a costly reconfiguration overhead both in execution-time and energy consumption. For example, the reconfiguration latency for a RU with 10% of the area of a Virtex XC2V6000 FPGA is 4ms (assuming that the frequency of the reconfiguration circuitry is 50 MHz). In addition, there exist an energy overhead due to the energy consumed in order to move the configuration from their off-chip storage to the reconfigurable device. Nevertheless, when analysing the performance and energy consumption of reconfigurable systems, it is often assumed that configurations are already loaded. This kind of analysis only depicts ideal results and they may lead to wrong

conclusions since the penalizations introduced during the reconfiguration process may have an important impact on the actual system performance.

1.4. Our Contributions

The target of this work is to develop a set of techniques to manage the dynamic reconfigurations while minimizing as much as possible the reconfiguration drawbacks. Our approach prevents that the reconfiguration latency degrades the system performance and, at the same time, it attempts to reduce as much as possible the energy overhead introduced by the reconfigurations. To this aim we have developed a set of scheduling techniques and we have proposed a new configuration memory hierarchy. The same techniques can also be used in an ASIP context.

We have designed a prefetch module that attempts to load configurations in advance hiding the loading latency behind real task execution. Therefore, those configurations that can be pre-fetched do not introduce any execution time overhead. Our prefetch module obtains good results while introducing an affordable run-time penalty due to its computations. Apart from the prefetch module we have also designed other two configuration optimization modules, namely, reuse and replacement modules. The aim of the reuse module is to identify, within the configuration scheduling, which configurations have to be loaded to be executed, and which can be directly executed because they were loaded in previous iterations, and they have not been removed to load another configuration. The main goal of the implemented replacement technique is to reduce the number of RU reconfigurations

reusing those tasks that were previously loaded. To do this we have implemented a replacement heuristic based on a well-known memory-page replacement strategy that when a new task must be loaded, it replaces the configuration that is going to be requested farthest in the future. These two modules present also good results combined with the prefetch module.

Typically, the configuration memory hierarchy for reconfigurable HW is composed of a reconfigurable fabric that stores the configurations that are ready for execution and an off-chip memory where the remaining configurations are stored. This scheme is usually present on fine-grain architectures, as FPGAs. One interesting improvement, often introduced for coarse-grain devices, consist in adding a smaller intermediate on-chip configuration memory, where the configurations of the running tasks are stored. This configuration memory is critical for the system, not only for the heavy configuration traffic required by dynamic applications' execution, but also for its energy consumption.

In this thesis we propose a heterogeneous on-chip configuration memory layer, which will replace the typical homogeneous off-chip memory for fine-grain systems. This memory layer consists of two memory modules, one optimised for High-Speed (HS), and another optimised for Low-Energy (LE). Hence, we propose a configuration memory hierarchy that provides fast reconfiguration, and at the same time the possibility of achieving energy savings whenever this speed is not needed.

In order to optimise the potential advantages of this heterogeneous memory layer, we have developed three different mapping algorithms that have been included in the scheduling system. The systematic mapping algorithms analyse at design-time

the features of the tasks that compose an application and interacts with the prefetch module. These modules must decide whether configurations should be stored in the LE, in the HS memory, or if it is necessary in no one of them, loading them into the reconfigurable resources form the external memory. Storing a configuration in the LE memory reduces the energy reconfiguration overhead but at the cost of a possible increase in the execution-time. The goal of our mapping algorithm is to identify a partition of the configurations that minimise the reconfiguration energy overhead without increasing the execution-time overhead significantly. In order to conveniently map the configurations to the different memories is very important to identify which are those tasks that have a high impact in the whole system performance, and assign those tasks to the HS memory whereas the remaining tasks are stored on the LE memory.

This proposal has been evaluated for fine-grain and coarse-grain platforms. In the case of fin-grain platforms we have used the data from the XILINX Virtex series FPGAs [Xili09] since currently they are clearly dominating the fine-grain reconfigurable market. For the coarse-grain platforms we have selected an architectural simulation environment for coarse-grain architectures called CRISP (Configurable and Reconfigurable Instruction Set Processor) [MMSY07]. CRISP is an academic simulator that can be used to simulate customized coarse-grained reconfigurable instruction set processors. The power of this architecture lies in the reconfigurable logic, which is composed of complex blocks, and is divided in independently enabled slices in order to reduce overall energy consumption and reconfiguration times. Also important is the tight coupling to the main microprocessor

(the reconfigurable logic is seen as an extra functional unit) that allows quick control and data communication between the processor and the reconfigurable logic.

This work has also been extended further in order to include extra heterogeneous memory levels in the configuration memory hierarchy. This new architecture provides more possibilities for the mapping algorithm, because the configurations are classified in different levels taking into account how critical they are for the application executions and each configuration is stored in the corresponding memory level.

1.5. Motivational Example

As we have previously mentioned multimedia is one of the fields where applications possess more dynamic behaviour. New multimedia standards, like MPEG-4, are object oriented and they support building heterogeneous applications where 3D-object, natural images, video and web elements can be displayed at the same time. In addition, the number of objects and their type may fluctuate very fast at run-time, and the platform must adapt itself to all these changes.

For instance, in 3D games new objects may appear at any point of time, and since they partially depend on the player's movement, it is impossible to predict at design-time when they will be displayed. Hence the platform must face this dynamic behaviour at run-time. To display a new object, the platform may have to decompress the object information, carry out a computing intensive rendering process, apply some specific shading and texturing improvements, and finally perform a rasterization

phase. If we assume that each phase is carried out by a different task, at least five tasks will be needed as we can see in Figure 4, and the platform must execute them meeting the exigent timing constraints. The figure depicts a task graph example including the execution time of each one of these tasks (Tex).

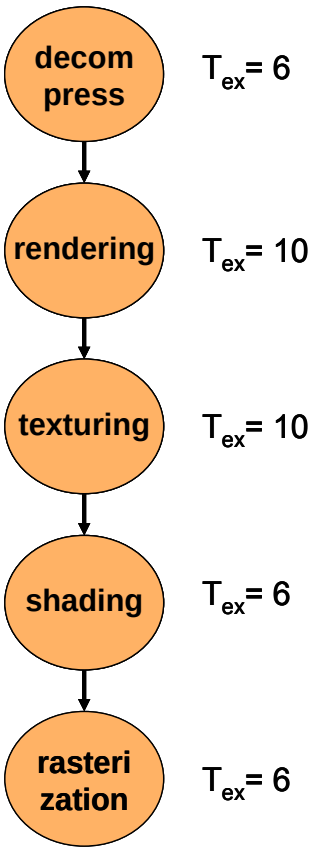


Figure 4. Tasks for displaying a 3-D object.

Although some of these tasks could be executed as SW processes, and only the most time consuming tasks as HW processes, we will implement all the tasks on Reconfigurable Units (RU) in order to obtain the maximum performance. Figure 5 shows a first approach to this problem, where each task is executed on a different RU. In this case there are as many RUs as tasks, but usually the number of available RUs is smaller than the number of tasks in the system since one of the main advantages of reconfigurable hardware is that the same RU can be reused for

several tasks. Moreover, in the figure the reconfiguration time associated to each task is not considered, assuming that each RU has the proper configuration already loaded. Hence it can start carrying out the computations as soon as the data are ready.

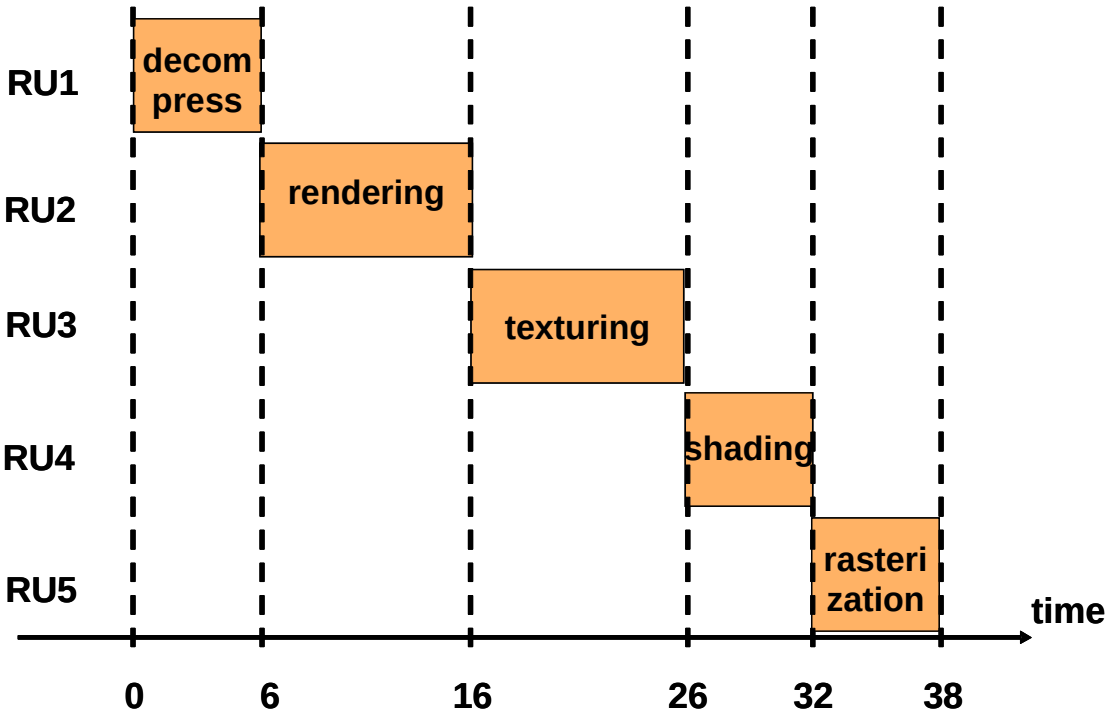


Figure 5. Example of task execution without reconfigurations.

Figure 6 presents the same execution sequence but taking into account the delays due to the run-time reconfigurations. In this example the reconfiguration latency is four time units.

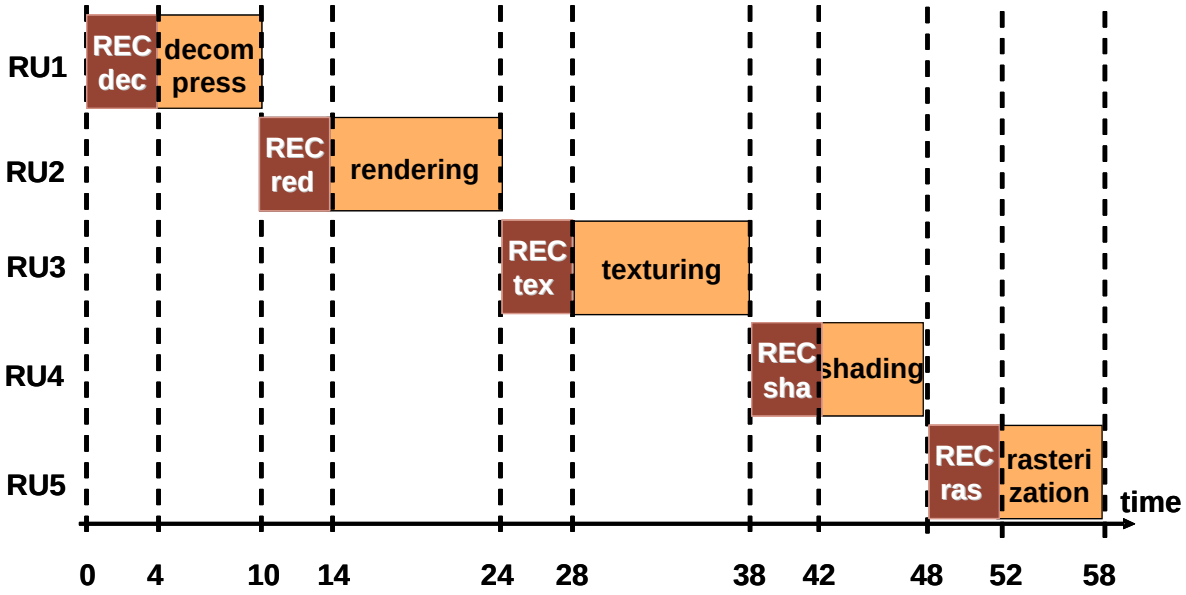


Figure 6. Example of task execution with reconfigurations. REC: reconfiguration.

In Figure 6, all the reconfigurations were introducing delays in the system execution. However, as soon as the system knows that a task graph must be executed, and identifies which reconfigurations must be carried out, it can apply a prefetch technique that will attempt to carry out the reconfigurations in advance. This option is depicted in Figure 7. In this case, as the execution time of each task is higher than the reconfiguration time, all the reconfigurations, except the corresponding to the decompress task, can be hidden. As this solution dedicates one RU to each task it provides the best possible performance, but obviously with a high HW cost.

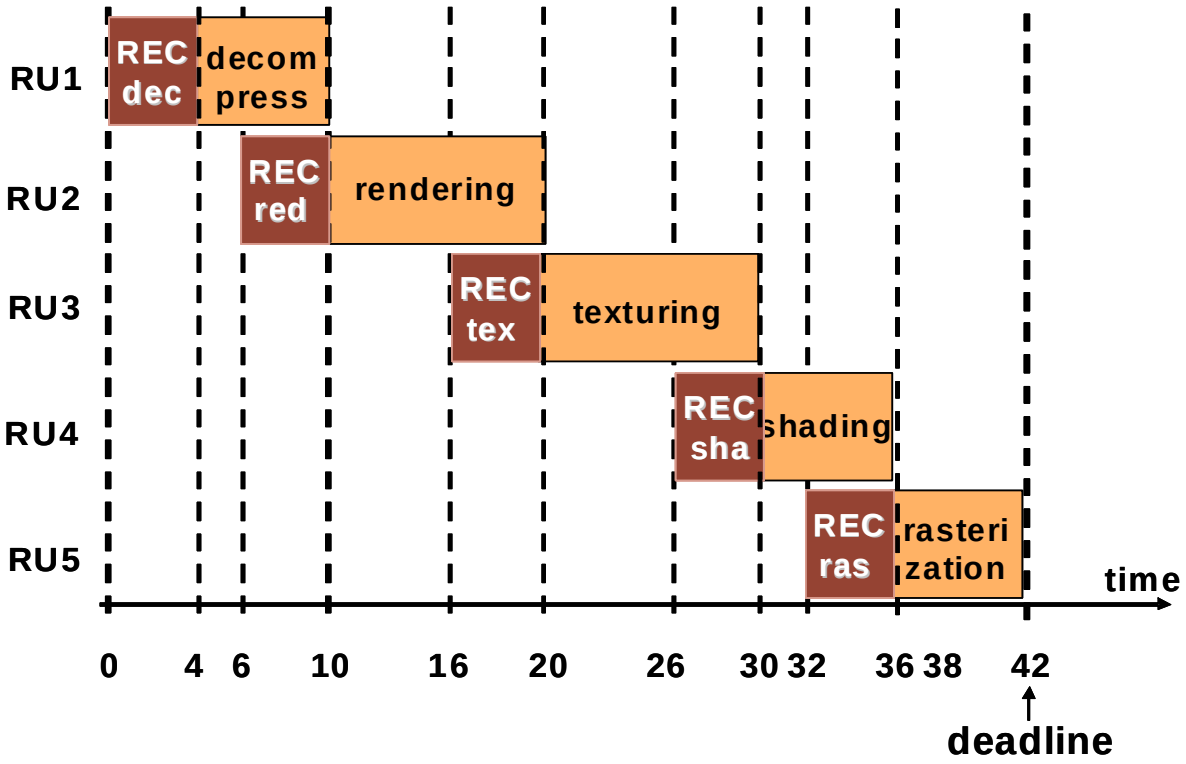


Figure 7. Example of task execution with reconfigurations and applying a prefetch approach (REC: reconfiguration).

On the other side, if we try to reduce as much as possible the HW cost, the system can take advantage of the run-time reconfiguration, and all the tasks could be executed in just one RU. However, as can be seen in Figure 8, in this case the reconfiguration latencies cannot be hidden. And if, for example, a deadline of 42 time units is established it cannot be met.

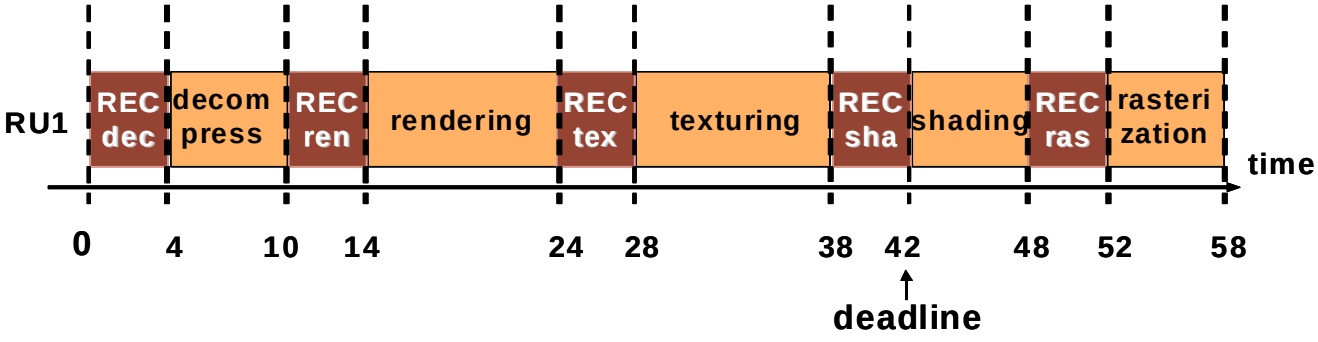


Figure 8. Example of task execution with just one RU (REC: reconfiguration).

Between these two solutions exist several intermediate possibilities, with different trade-offs between execution time and HW cost, and even power consumption. For example, if our platform has only 3 RUs, since the system knows that these five tasks must be executed and also knows that they must be loaded through a reconfiguration process, it can schedule these reconfigurations in advance, with a prefetch approach, if there are enough reconfigurable resources available, overlapping the reconfiguration of a task with the execution of the previous tasks. The benefits of this approach are depicted in Figure 9. In this case, the system has hidden the latency of four of the five reconfigurations, since they have been loaded in advance. Hence, only the task decompress introduces a delay and the system obtains a similar performance than the system that included five RUs.

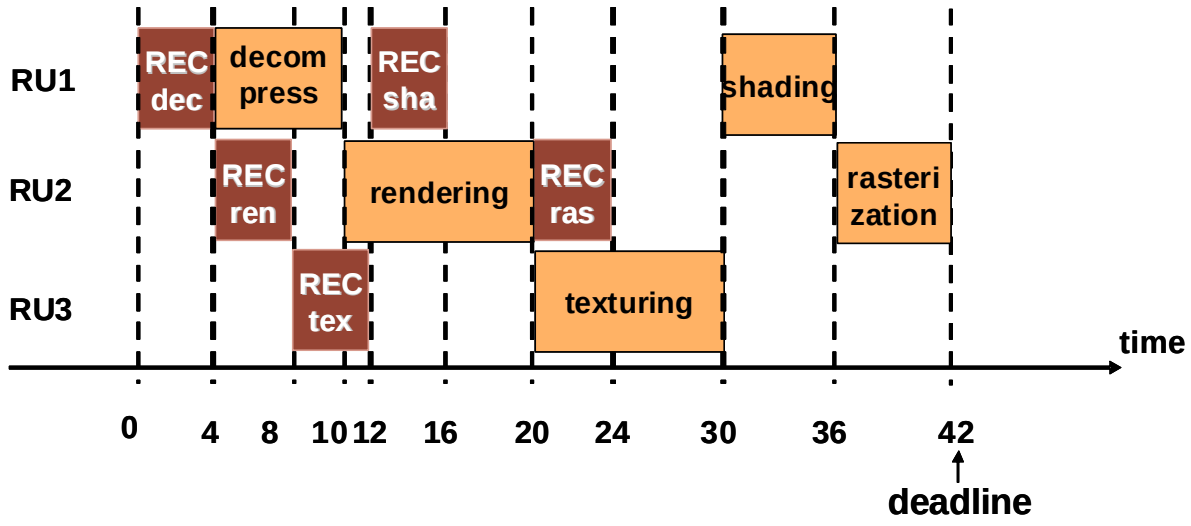


Figure 9. Example of task execution applying a prefetch approach with 3 RUs (REC: reconfiguration).

In order to further optimise the reconfiguration process, the system may keep a record with all the configurations that are currently loaded. Each time the system receives the order to execute a set of tasks, it will check this record identifying if the

configuration of any of these tasks is already loaded. In this case, that configuration can just be reused. Figure 10 exemplifies this case assuming that this set of tasks is executed twice consecutively and in the second execution the task texturing can be reused without reconfiguration.

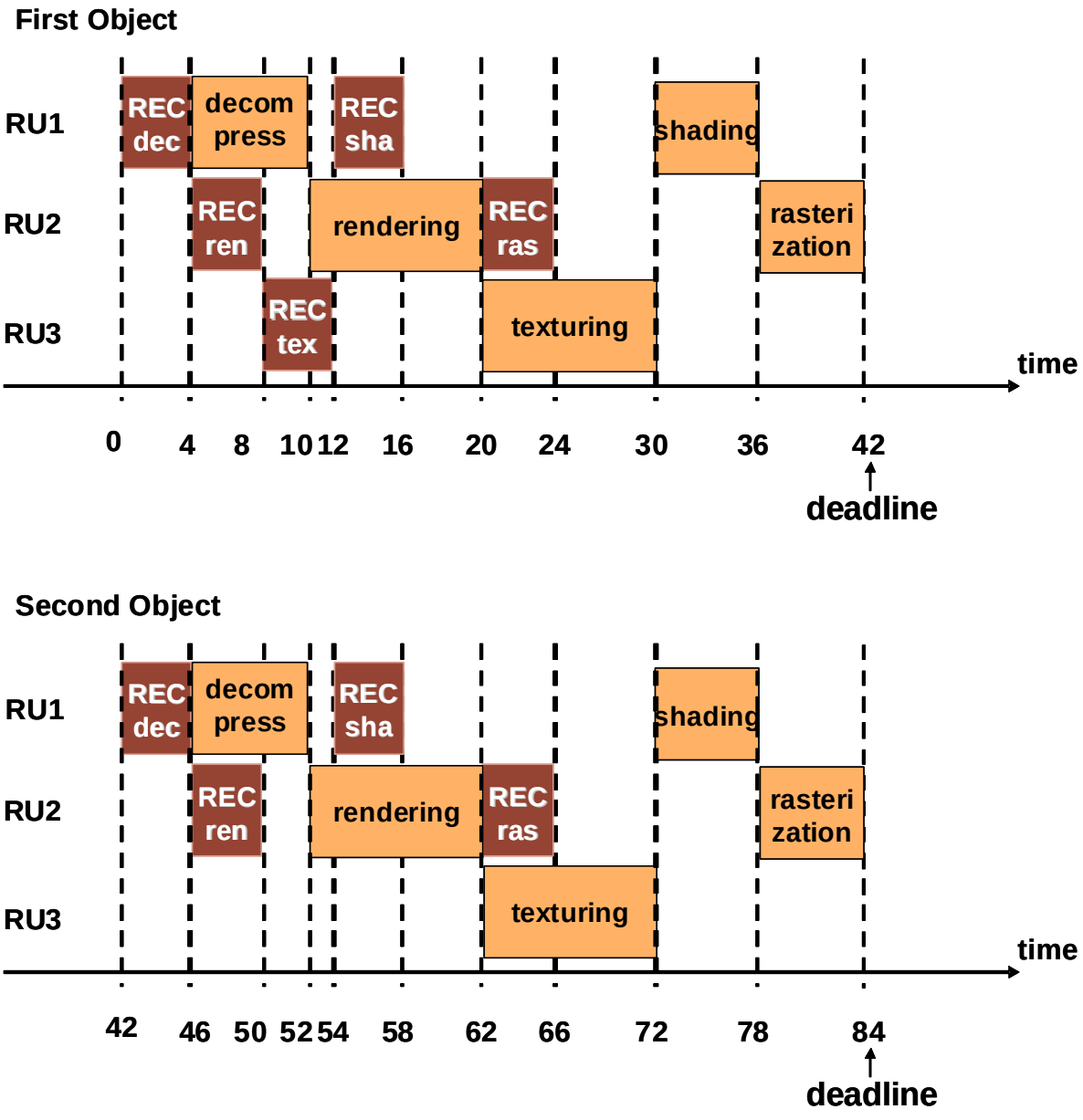


Figure 10. Example of task execution applying prefetch and reuse approaches (REC: reconfiguration).

Compared to the previous case, there are no execution time reductions, since the latency of the reconfiguration of the texturing task was already hidden by the prefetch approach. However, the reconfiguration energy overhead has been reduced 10%, as we have eliminated one of the ten reconfigurations.

These results can be improved if the system applies a replacement technique that assigns more priority to the decompress task than to the others. The reason for this higher priority is that the reconfiguration of this task is the only one that cannot be hidden applying a prefetch approach. If the system assigns more priority to this task, probably, the second time that a similar object must be decompressed this task will be reused, and the reconfiguration latency will virtually disappear, like it happens in Figure 11.

In this example the system does not replace the task decompress in order to load another task. Hence, if this set of tasks is executed multiple times consecutively, decompress will be loaded just the first time, and will be reused the remaining executions.

Although the initial reconfiguration overhead was very high, this example depicts how it can be reduced until it becomes negligible if the appropriate support is included. The main focus of these previous techniques is to reduce the execution-time overhead. However, the reconfiguration process also introduces an important energy overhead and it would be very interesting to find a way to reduce it.

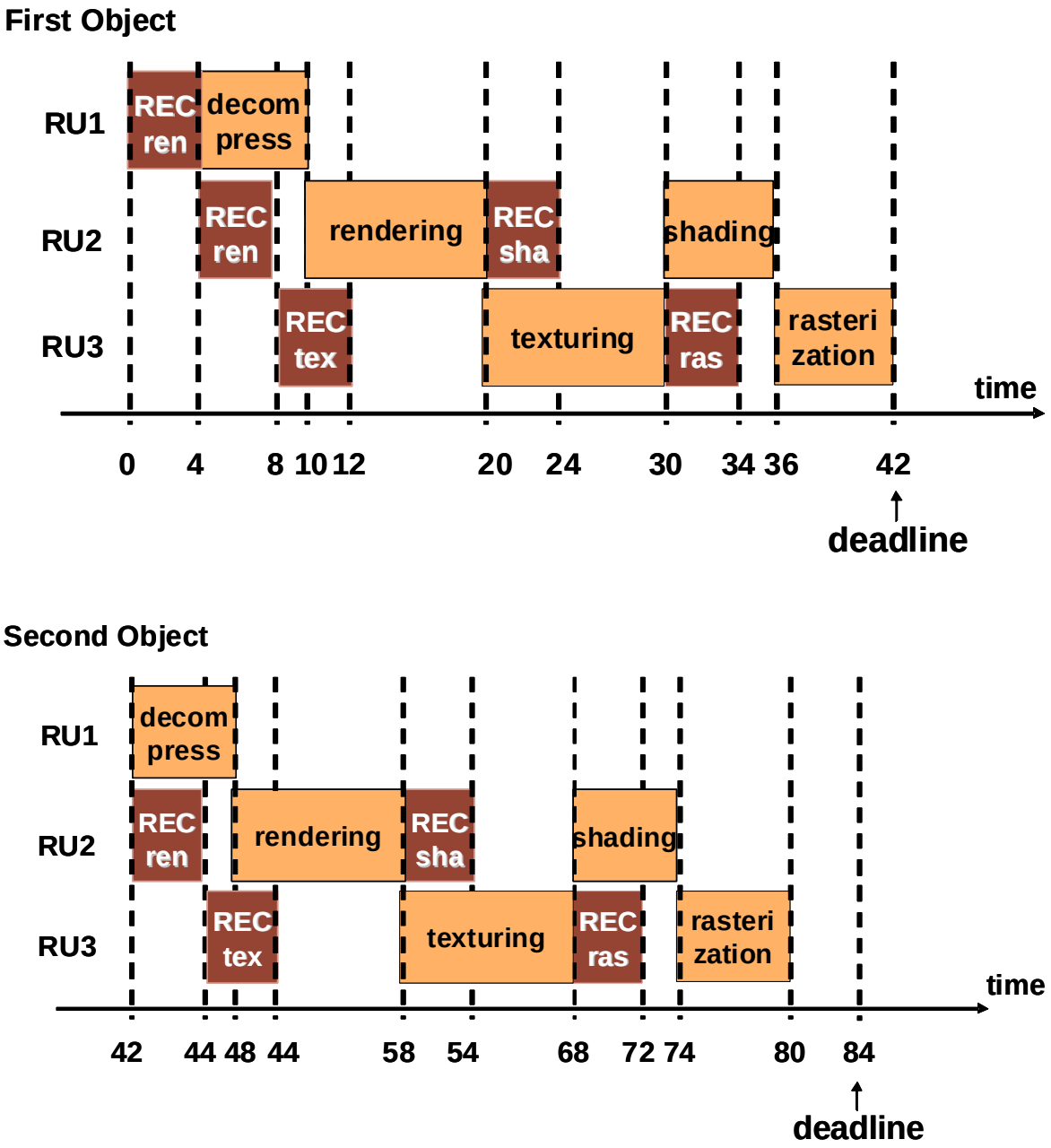


Figure 11. Two consecutive executions of the task graph applying an optimal replacement approach (REC: reconfiguration).

To this end we propose to include an intermediate on-chip configuration memory layer that will store the configurations of the active tasks. This idea is similar to the inclusion of an instruction cache in conventional processors and has already been applied in some coarse-grain reconfigurable architectures.

In recent years extensive efforts have been focused on reducing SRAM memories energy consumption. As a result, a new type of memories is currently available in the market oriented to Low-Energy, with similar features than the High-Speed ones, but with worse speed ratios. Different memory manufacturers, for example, Virage Logic [Vira09] and Micron Technology [Micr09], have introduced some of these innovating techniques in the design of Low-Energy SRAM memory.

Consequently, the embedded systems designers must select the appropriate memory for their platform among the wide number of possibilities available on the market. However, selecting a memory optimized for high-performance, usually involves energy consumption overheads, while selecting a memory optimized for reducing the energy consumption may lead to important performance degradation (more data-path cycles are needed per access). Figure 12 depicts the execution profile using a memory optimized for Low-Energy to implement this on-chip configuration memory instead of the High-Speed one. This memory is 50% slower (each reconfiguration lasts 6 time units) but it saves 30% more energy per access than the one optimised for performance.

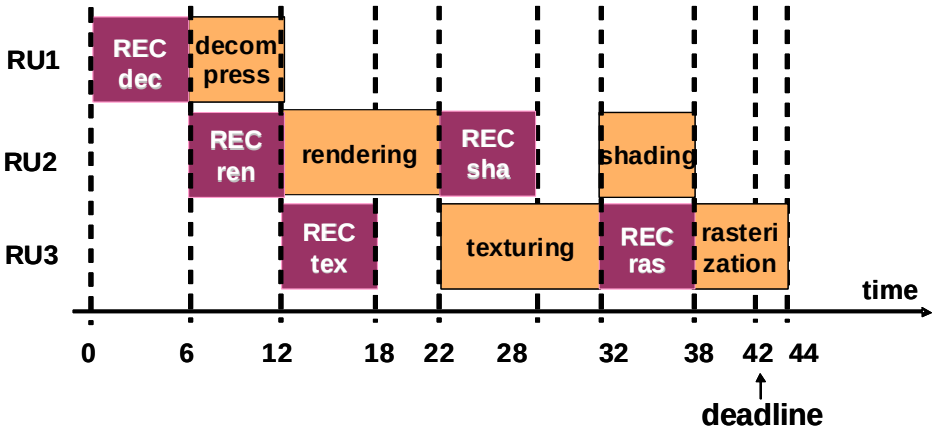


Figure 12. Example of task execution with one Low-Energy configuration memory (REC: reconfiguration).

As we can see in this figure, selecting a memory optimized for reducing the energy consumption may lead to important performance degradation. Since designers need both high performance and low energy consumption features, we propose to include two different types of memories in the configuration memory hierarchy, one optimized for High-Speed and the other one optimized for Low-Energy. Hence, we are potentially supplying high performance and low energy features to the configuration memory hierarchy. If the system manages to distribute the load of tasks between these two memories properly, moving selected tasks from the High-Speed memory to the Low-Energy one, important energy savings can be obtained without reducing the overall system performance. Figure 13 depicts an example of a schedule obtained with this configuration memory hierarchy.

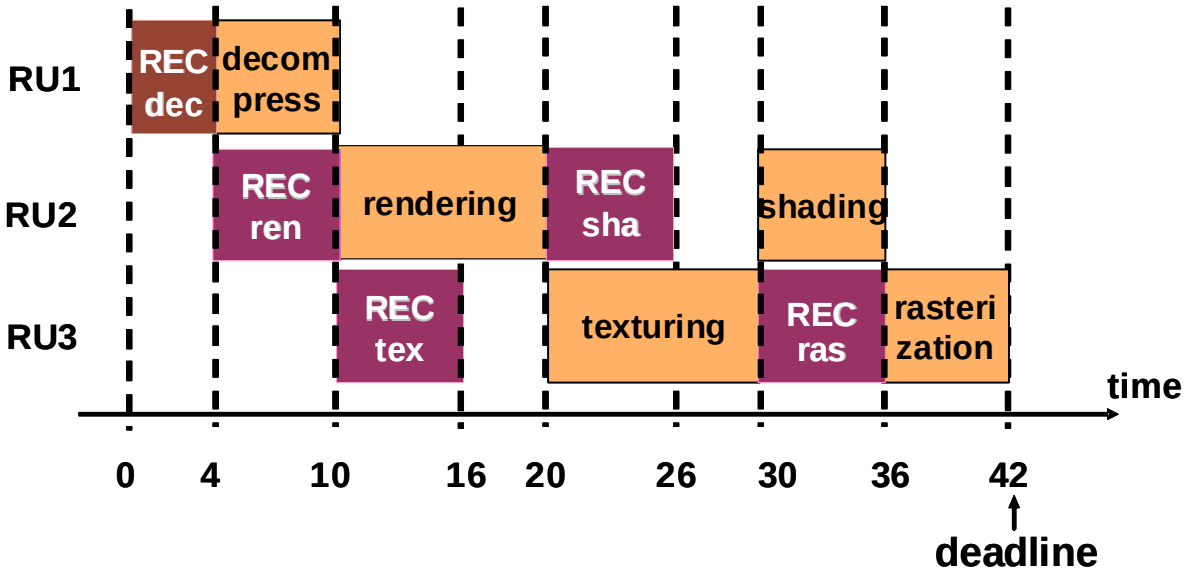


Figure 13. Example of task execution with two configuration memories one High-Speed and other Low-Energy (REC: reconfiguration).

As we can see in this figure, energy consumption has been clearly reduced since four tasks have been mapped to the Low-Energy memory while the performance level is kept constant.

Adding this extra on-chip configuration memory level to the system increases the design space exploration options because now the system must decide which configurations are stored in the High-Speed memory and which ones are stored in the Low-Energy memory. Hence, we have developed a mapping technique to make these decisions considering the interactions with the prefetch scheduler.

Chapter 2:
Current Reconfigurable HW Features

This chapter does not provide an extensive description of the available reconfigurable HW architectures, but only analyzes some important features that have a direct influence in the reconfiguration overhead, namely: the configuration granularity (coarse-grain vs. fine-grain), the number of configurations (or contexts) that can be loaded at the same time for a reconfigurable unit (single context vs. multi-context), and the type of reconfiguration (total vs. partial).

Detailed information over other considerations about reconfigurable HW design can be found in [SVKS01] and in [CoHa02].

We start this chapter describing in detail these features, and finally we describe three major examples of reconfigurable HW platforms.

2.1. Granularity

As we have mentioned before, reconfigurable HW devices are integrated circuits whose functionality can be changed through some control point configuration. The complexity of the configurable blocks, which enable that the device functionality can change, varies considerably from some system to others. This reconfigurable HW feature is called configuration granularity.

Reconfigurable HW is typically classified in two main categories regarding the configuration granularity, namely fine-grain and coarse-grain.

2.1.1. Fine-grain

In fine-grain architectures a configuration can specify the system behaviour at bit level. That is, different operation can be assigned to each system bit.

Hence, in fine-grain architectures, the logic operations are mainly done at the bit or small word (<= 4 bits) level, providing a very flexible architecture that can be customized to the specific needs of an application. For instance, if an application only needs a 6-bit adder for a particular operation and a 23-bit adder for another, the designer does not need to use a 32-bit adder to perform these lower precision computations. Instead of that, the designer can directly implement an adder with the specific size requested by each operation.

FPGAs are the most representative architecture for fine-grain approaches, and currently they are clearly dominating the market. The main reason is that they are

based on a mature technology with many years of development, and supported on a wide set of design tools.

The basic architecture of FPGAs consists of three kinds of components: computational logic blocks (CLBs), routing, and input/output (I/O) blocks. Each CLB typically contains two or four identical programmable slices. Each slice usually contains two programmable cores with few inputs (typically four inputs) that can be programmed to implement a 1-bit logic function. Programmable routing surrounding CLBs ensure the communication among them, while programmable I/O blocks connect the FPGA with the system peripherals. Finally, specific I/O ports are employed to download the configuration data from the external configuration memory.

Above all, fine-grain architectures provide the highest possible flexibility to the designers to create the optimum configuration for each algorithm.

However, due to their configuration and general-purpose nature, fine-grained reconfigurable systems suffer by a number of drawbacks, which become more important when they are used to implement word-level units and datapaths.

Fine-grain architectures present lower performance and higher power consumption than ASICs ([PTCC09], [PTMM09]). This happens because word-level modules are built by connecting several CLBs causing performance degradation and power-consumption increase. To build a word-level unit or datapath a large number of CLBs must be interconnected using a large number of programmable switches resulting in huge routing overhead and poor area utilization. In fact, up to 80%-90% of the area of commercial FPGAs is used for routing purposes.

Another important drawback is the large reconfiguration time. The configuration of CLBs and interconnections wires is performed at bit-level by applying individual configuration signals for each CLB and wire. This results in large configurations that have to be downloaded from the configuration memory, which usually is off-chip, with the consequent large reconfiguration latencies. Hence, the reconfiguration latency may degrade the performance if the system carries out frequent reconfigurations.

2.1.2. Coarse-grain

As we have just mentioned, fine-grain systems exhibit low/medium performance rates, high time and energy reconfiguration overheads, which become pronounced when, they are used to implement processing units and datapaths that perform word-level data processing. The research efforts towards reducing the delay, energy, and/or area cost of fine-grained reconfigurable HW have led to a number of reconfigurable logic devices called coarse-grain reconfigurable devices.

Instead of providing configurability at the level of individual gates, flip-flops or look-up tables, the coarse-grained architectures often provide arithmetic logic units and other larger HW blocks, like register banks or RAM memories, that can be combined to perform the demanded computations. Coarse-grain devices implements word-level operations and special-purpose interconnections retaining enough flexibility for mapping different applications onto the system. Hence, these systems are application domain-specific systems, whose reconfigurable logic and interconnections are configured at word-level. Due to their coarse-grain granularity,

when they are used to implement word-level operators and datapaths, coarse-grain reconfigurable systems offers higher performance, reduced time and energy overhead, better area utilization, and lower power consumption than the fine-grain ones.

Typically, the coarse-grain reconfigurable HW is tightly coupled to a processor. The coarse-grain reconfigurable part undertakes the computationally-intensive parts of the application, while the processor is responsible for the remaining parts.

Concerning the interconnection network, it consists of programmable interconnections that ensure the communication among RUs. The wires are grouped in buses each configured by a single configuration bit instead of applying individual configuration bits for each wire as it happens in fine-grain systems.

An important consequence of the coarse-grain granularity is that the footprint of the configuration storage is significantly smaller than in the fine-grain architectures because both the HW blocks and the interconnections are configured at word-level. This leads to reductions of one or two order of magnitude in the final configuration footprint.

In addition, since the configurations are relatively small, it is possible to store some of them on-chip achieving further reconfiguration-latency reductions.

However, the development of coarse-grain reconfigurable systems is a trade-off between flexibility and circuit specialization. Flexibility means the capability of the system to adapt and respond to the new requirements of the applications implementing circuits and algorithms that were not considered during the system's

development. According to the flexibility coarse-grain reconfigurable systems can be classified in two categories: application-domain specific and application-class specific systems [VaSo07].

On the one hand, an application-domain specific system targets a set of applications of a certain domain. It consists of specific HW blocks and reconfigurable interconnections, which are based on domain needs, properly organized to retain flexibility for implementing efficiently the required circuits. However, due to this flexibility, the design complexity increase and the efficiency decrease. The vast majority of the existing coarse-grain reconfigurable systems belong to this category.

On the other hand, application-class specific systems are flexible ASIC-like architectures that have been developed to support only a limited and predefined set of applications. Hence, they only support these specific similar configurations. They consist of specific types and number of processing units and particular direct point-to-point interconnections with limited programmability. The reconfiguration is achieved by loading one of the predefined configurations. Thus, these systems do not meet one of the fundamental properties of reconfigurable systems, namely the capacity to support functionality upgrades and future applications. They are the two ends of the wide range of possible coarse-grain architectures available in the market.

Although coarse-grain reconfigurable architectures offer a very high degree of parallelism to improve performance in data-intensive applications, a major bottleneck arises because of the large memory bandwidth required to feed data concurrently to the underlying processing units. In addition, when frequent reconfigurations are demanded, loading the new configurations may intensify the bandwidth bottleneck.

Typically, coarse-grain architectures attempt to reduce this problem by including a specific configuration-memory hierarchy.

The techniques developed in this thesis can be used for both fine-grain and coarse-grain platforms. The main difference is that the number of bits needed to configure a task is bigger for fine-grain than for coarse-grain (this difference varies a lot taking into account the flexibility level allowed by each coarse-grain platform). Thus, the reconfiguration latency in fine-grain platforms is bigger, and hence the necessity of including a dedicated support to reduce the reconfiguration penalization is clearer and, at the same time, more difficult.

In any case, for most commercial coarse-grain architectures the reconfiguration latency is still important enough to include specific support to minimize it. For example, the size of a configuration in the RAP platform [Elix09] is 15Kb, and in the DAPDNA-1 [lpfl05] platform is 5Kb. If an 8 bit-width bus is used to write these configurations, with a writing frequency of 50 MHz, this operation will consume 0,3 and 0,1 ms respectively, although these latencies are quite far away from the 4 ms needed to reconfigure only 10% of a Virtex-2 XC2V6000 in the same conditions, they can still create an important penalization, specially when they carry out reconfigurations frequently.

2.2. Context

In current reconfigurable HW the configuration bits are distributed in memory elements along the platform in order to customize each configurable element. To change the functionality of a reconfigurable HW platform it is necessary to change the bits loaded in these memory elements.

Single-context platforms only stores in these on-chip distributed memory-elements one configuration for each element, and if another configuration is needed, a reconfiguration circuitry is used to read the new configuration from an external memory and overwrite the previous one. Hence, in conventional single-context reconfigurable-HW only one active configuration is present at a given point of time.

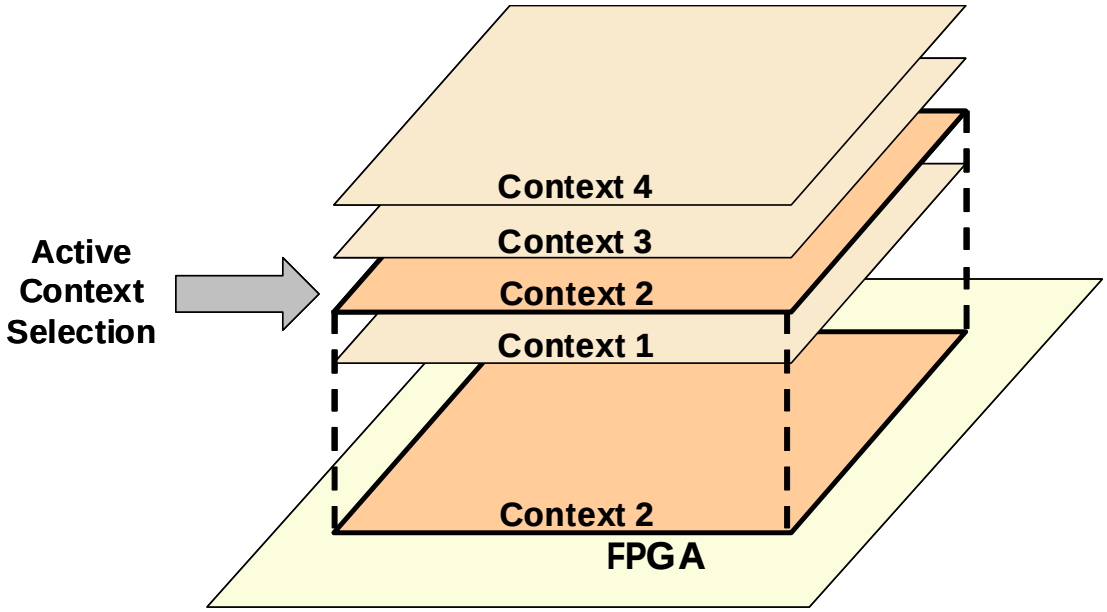


Figure 14. Multiple-context FPGAs.

To reduce the reconfiguration latency, multiple-context FPGAs ([TCJW97], [LPPA02]) store several active configurations at the same time (frequently called

contexts). For example in [LPPA02] the authors propose a FPGA architecture that stores four configurations for each element. The actual configuration that is executed at a given point of time is selected using extra control logic as can be seen in figure 14.

The actual context is not selected individually for each element but it is globally selected for the whole FPGA in order to simplify the control logic. To change from one context to other the platform only needs one clock cycle since the configuration information is already loaded. Hence, as long as the FPGA only executes the configurations stored in the contexts, the reconfiguration latency will be only one cycle.

Unfortunately, including multiple contexts in an FPGA entails an important extra area cost, firstly because the size of configurations storage is multiplied by the number of included contexts, and secondly, because extra logic must be added to select the context. Due to this area cost multi-context FPGAs have only been proposed in academic environments.

However, this idea has been successfully implemented in coarse-grain platforms because they need less configuration bits. Hence, the extra cost due to the inclusion of several contexts is much more affordable. Thus, some commercial architectures that work with several contexts exist, like MorphoSys [SLLK00] or RAP [Elix09].

The work developed in this thesis is mainly focused on single-context platforms because in these platforms the reconfiguration latency is especially important, and

they clearly need specific support to reduce the penalizations due to the reconfigurations. Besides, currently these architectures widely control the market. The extension to multi-context platforms is however quite feasible but it is not worked out in this thesis.

2.3. Partial and Global Reconfiguration

As we have mentioned in the previous section, in multi-context FPGAs the same active configuration must be selected for the whole FPGA. Hence, a region of the FPGA cannot execute Context 1 while other region executes Context 3. In this kind of platforms, if you want to modify the configuration of a certain region you need to load a new configuration for the whole platform. This process is normally called global reconfiguration. On the contrary, in some FPGAs it is possible to change the configuration of certain FPGA region without modifying the remaining FPGA area. This process is called partial reconfiguration. Partial reconfiguration provides two important advantages. Firstly, it allows managing independently the simultaneous execution of several tasks. Secondly, it directly reduces the configuration latency, because in order to execute a task in a certain region, the system only needs to load the configuration bits of that region, whereas in global reconfigurable platforms all the configuration bits must be loaded.

One drawback of partial reconfiguration is that, in order to implement it, it is necessary to work with addresses that indicate where the configuration has to be loaded. In global reconfigurable platforms these addresses are not needed because

there exists only one way to carry out the configuration. However, using addresses is a really small drawback, if it is compared with the extended opportunities that partial reconfiguration opens.

Some partial reconfigurable platforms examples are the XILINX families of FPGAs, Virtex-4, Virtex-5, Virtex-6, Virtex-II and Virtex-II Pro [Xili09], the Atmel family AT40K [Atme07] and coarse-grain platform Chimaera [YMHB00], PipeRench [CWGS98] and NAPA [RLGC98].

The work developed in this thesis is mainly oriented to partial reconfigurable HW. We have adopted a model of partial reconfigurable HW where the reconfigurable resources are divided in a set of basic units where each one includes a specific support for the communications since this is a common assumption in reconfigurable systems.

Since the appearance of the first partially reconfigurable FPGA, several research groups have considered as necessary to divide the reconfigurable resources in a set of independent and encapsulated units and to add a specific support to manage the communication between tasks. The first model for dynamic reconfigurable HW with these features was present in [Breb96] where each RU was called SLU (Swappable Logic Unit). A task could be loaded in each SLU, and all these tasks could be executed in parallel. The main reason to partition the reconfigurable HW into reconfigurable symmetric units is that this approach prevents the need to carry out complex routing computations before loading a new configuration.

A little bit afterwards [MeLJ98] proposed a similar model. More recently several research groups have proposed other similar approaches, as for example [NoBa04b], [LeMi03], [StWP04], and [RMMS08] that also divide a reconfigurable platform into a set of reconfigurable units.

Now we present three examples of major reconfigurable platforms that combine different features for the previously discussed list:

- Virtex-5 FPGAs: Fine-grain, single context and partial dynamic reconfiguration.
- CRISP: Coarse-grain, single context and partial dynamic reconfiguration.
- MorphoSys: Coarse-grain, single context and partial dynamic reconfiguration.

Both CRISP and MorphoSys are coarse-grain, single context and partial dynamic reconfigurable platforms. The main difference between them is that CRISP has been used mainly in academic environments up to now, and MorphoSys has being implemented commercially. However, the FEENECS/COFFEE environment which is the follow-up of CRISP has been used for exploring internal wireless designs at IMEC, such as the Syncpro chip [RJLA09].

2.3.1. Virtex-5 FPGA family

Nowadays FPGAs not only includes CLBs, routing, and input/output blocks, but also a balanced mix of performance-efficient components as on-chip memories,

multipliers or operational units and in some cases small processors. Hence, at present the division between fine-grain and coarse-grain is more and more diffuse. An example of these modern FPGAs is the Virtex-5 FPGA family.

Virtex-5 FPGAs are built on 65-nm triple-oxide technology using advanced silicon modular block architectures and providing additional levels of system integration. This new Virtex FPGA family presents four different platforms or families with different system features:

- The LX platform optimized for high-performance logic applications.
- The LXT platform, optimized for high-performance applications logic with advanced low-power serial I/O.
- The SXT platform optimized for high-performance arithmetic- and memory-intensive DSP (Digital signal processing) with advanced low-power serial I/O.
- The FXT platform, optimized for embedded processing of very high-speed serial I/O.

Compared to the previous Virtex-4 family, Virtex-5 devices provide 30% higher average speed and 65% higher capacity in the largest device whereas the dynamic power consumption has been reduced by 35% and chip area is 45% smaller.

Since FPGAs were introduced in the mid 1980s, the logic fabric for most of them has been based on the same fundamental four-input look-up table (LUT) architecture. The Virtex-5 family is the first FPGA platform to offer a true six-input

LUT fabric with fully independent, not shared inputs. Six-input LUTs have been incorporated in order to increase Virtex-5 performance and density. Each LUT can be configured as a six input function or as two functions of five inputs. Virtex-5 CLB scheme is depicted in Figure 15.

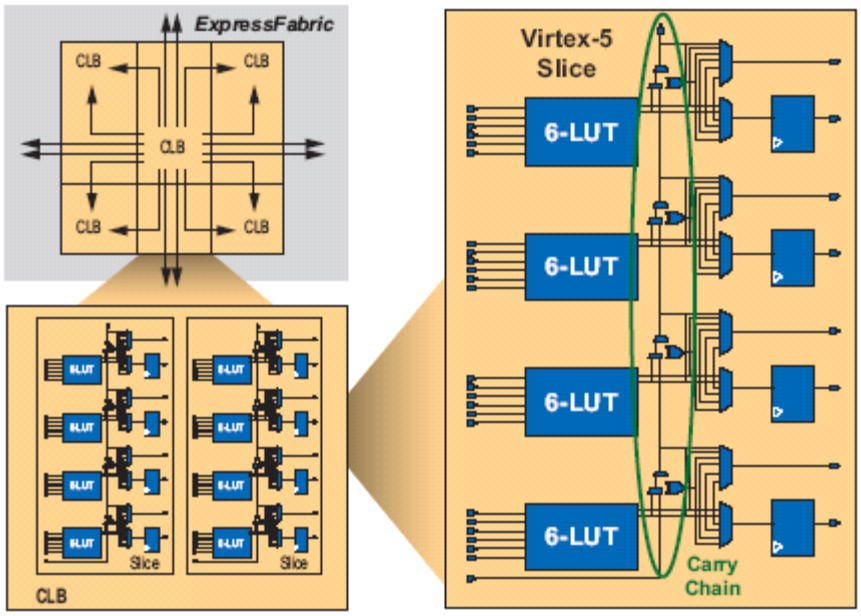


Figure 15. Virtex-5 CLB.

Synchronous dual-ported block RAM is also an important advance introduced at Virtex-5 FPGAs family. The size of each block RAM has been increased to 36 Kb, but it can also be used as two independent 18-Kb block RAMs. The data bus width is programmable from 1 bit to 36 bits. In simple dual port mode (one port write, one port read) the data bus width can be as high as 72 bits, effectively doubling the data bandwidth. The unused 18-Kb blocks can be turned off to save power, especially also static power if power gating is applied appropriately.

The block RAM has integrated FIFO control logic, simplifying the design of asynchronous (or synchronous) FIFOs running as fast as 550 MHz without consuming any logic resources.

The 72-bit-wide block RAM now includes 64-bit error checking and correction (ECC) control logic. Like the integrated FIFO support, the integrated ECC improves memory performance and eliminates the cost associated with traditional fabric-based solutions.

The Virtex-5 family also includes improvement in the DSP slices. They are now providing 25 x 18-bit multipliers, mainly for more efficient floating-point designs. These DSP48E slices can be directly cascaded for higher performance in digital filtering or video broadcast applications. Direct cascading also saves power – as much as 40% compared to competing solutions. Each DSP48E also include an optional adder, subtracted, and accumulator.

Virtex-5 family adds a phase-locked loop (PLL) to each clock management tile (CMT), which now contains two digital clock managers (DCMs) and one PLL. The CMT thus offers the best of both worlds: the robust versatility and precise incremental phase shift capability of a digital clock manager combined with the jitter reduction from the analog PLL. The largest device in the family has six CMTs capable of generating and manipulating 550-MHz clocks, supporting the performance of Virtex-5 logic and block functions.

Xilinx has also developed a new diagonal interconnect pattern for Virtex-5 FPGAs, to enhance performance by reaching more places in fewer hops. The new

pattern increases the number of logic connections achievable within two or three hops. These features are transparent to Virtex-5 FPGA users, resulting in easier routability and higher overall performance. In Figure 16, we can see this new interconnection pattern, where each box represents a CLB.

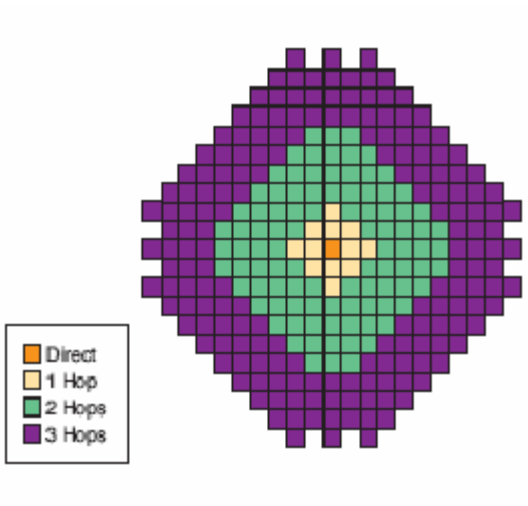


Figure 16. Virtex-5: Diagonally Symmetric Interconnect Pattern.

Currently System-on-a-chip platforms are a major trend in embedded systems. These systems usually include one or several processors, HW blocks, memory resources and several buses. Xilinx has provided the possibility to build these systems using Virtex FPGAs by including in the Virtex-II Pro and Virtex-4 series one or two embedded PowerPC 32-bit RISC processors. These industry-standard processors offer medium-high performance and a broad range of third-party support. In addition, for those FPGAs that do not include an embedded processor (also called a hard-core processor), Xilinx provided a customizable 32-bit soft-core processor called MicroBlaze [XiMB09] that can be implemented in any FPGA using some of the reconfigurable resources. Moreover, Xilinx has developed a specific system-on-a-chip

design environment called EDK (Embedded Development Kit, [XEDK09]). In this environment designers can easily develop a processor subsystem tailored to their requirements, selecting peripherals from an IP catalogue and creating custom peripherals to interface with the system I/O. Features such as MicroBlaze instruction acceleration support, configurable-size instruction and data caches and multiple processor instantiations, coupled with the possibility of including custom peripherals, can be used to build powerful platforms that provide performance levels beyond the capabilities of traditional processing architectures. An example of a MicroBlaze processor subsystem instantiation is depicted in Figure 17.

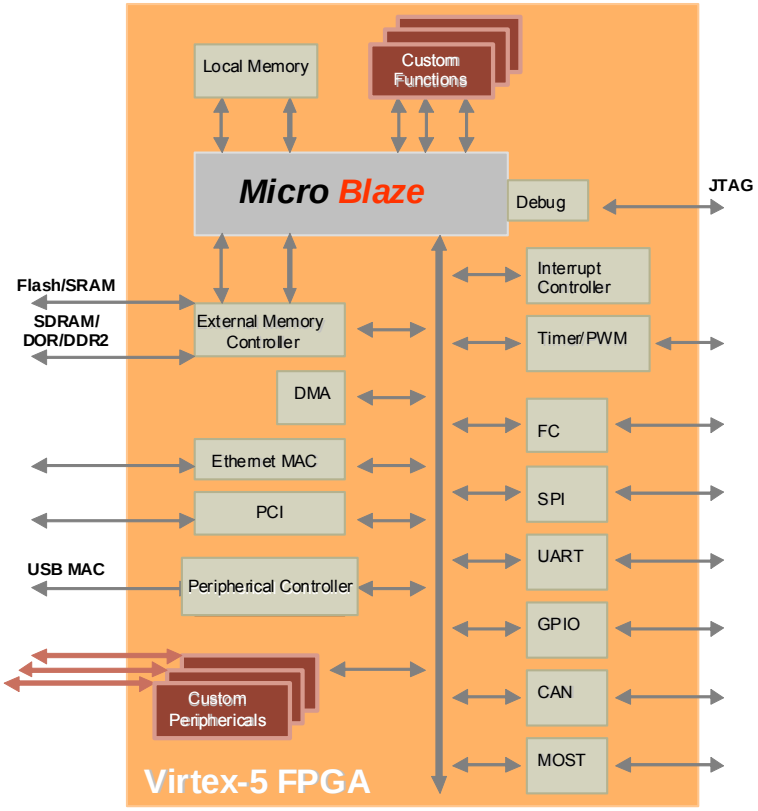


Figure 17. MicroBlaze processor subsystem design example.

As we have mentioned before, today digital designers face steep challenges. They have to meet aggressive performance goals, while reducing time-to-market to meet aggressive deadlines. To face this challenge Xilinx has developed a set of SW design tools that simplify the design process. Xilinx offers a collection of design tools that covers a wide range of the different designer's needs. This set of design tools goes from the most general-purpose evaluation and development platform to specialized design kits that offer support for low-cost embedded system evaluation and development (Virtex-5 ML505), or support for networking, telecom, servers, and computing systems design (Virtex-5 ML550), which allows designers to focus on their design without worrying about interoperability and standards compliance. Other Virtex-5 development tool kits provides pre-verified development solution to face parallel and serial PCI interface design challenges (Virtex-5 ML555), or comprehensive support to enable, build and verify robust high-performance memory interfaces. Xilinx also provides complete development platforms that evolve all the features of the above mentioned kits (The Xilinx Virtex-5 LX Development Kit, HiTech Global Virtex-5 PCI Express Development Platform, Nu Horizons Virtex-5 LXT Evaluation Kit [XilV-5]).

Beside this set of development tools for Virtex-5 FPGAs family, Xilinx provides the ISE Design Suite [XiSG09] that delivers a comprehensive solution for logic [XISE09], embedded [XEDK09], and DSP [XDSP09] design for all leading Xilinx FPGAs. In addition other companies, as Simplicity, have also developed design tools for Xilinx FPGAs.

The possibility of using all these mature design tools is one of the major reasons for the current FPGAs success. Other reasons are the inclusion of coarse-grain blocks and the improvement in the power and energy efficiency of the fine-grain reconfigurable devices. For all these reasons it is expected that the reconfigurable fine-grain market will continue growing in the following years.

2.3.2. CRISP

CRISP (Configurable and Reconfigurable Instruction Set Processor) is a simulation environment for coarse-grain architectures. In this environment designers can define a coarse-grain architecture and a memory hierarchy [BJVL03]. With these inputs CRISP generates a coarse-grained reconfigurable instruction set processor and a compilation framework. The main objective of the CRISP project is to accelerate multimedia applications in a power efficient way.

The original CRISP architecture is basically a main processor core tightly coupled to some coarse-grained reconfigurable logic. The coarse-grained reconfigurable logic is used to increase performance and decrease power consumption.

The power of this architecture lies in its reconfigurable logic, which is composed of complex blocks such as ALUs or multipliers, that operate on the data sizes typically found in multimedia applications (8 to 32 bits), and is divided in independently-enabled slices in order to reduce the overall energy consumption and reconfiguration times. From the point of view of the processor the reconfigurable-logic

blocks are just additional functional units (called RFU, Reconfigurable Functional Units) and just like any other functional unit, they can execute one operation every clock cycle reading and writing data from/to internal registers. This approach allows fast control and data communication between the processor and the reconfigurable logic.

Figure 18 presents the CRISP architecture. The main processor core reads its instructions from the level 1 instruction cache and obtains data via the level 1 data cache. Both caches are connected to a unified level 2 cache, which is connected to an external memory. The reconfigurable fabric, in the middle of the figure, is directly controlled by the main processor core, and contains a configuration memory that is loaded via the unified level 2 cache. This allows the reuse of configurations loaded from external memory and reduces reconfiguration times. The reconfigurable fabric can directly access the data cache via several data ports. Additionally, the reconfigurable logic communicates with the main processor core via a functional unit interface.

As shown in Figure 18, the RFU is divided in reconfigurable slices. These slices can be independently activated, and the number of active slices is carefully selected by a compiler in order to maximise the performance while reducing the energy consumption.

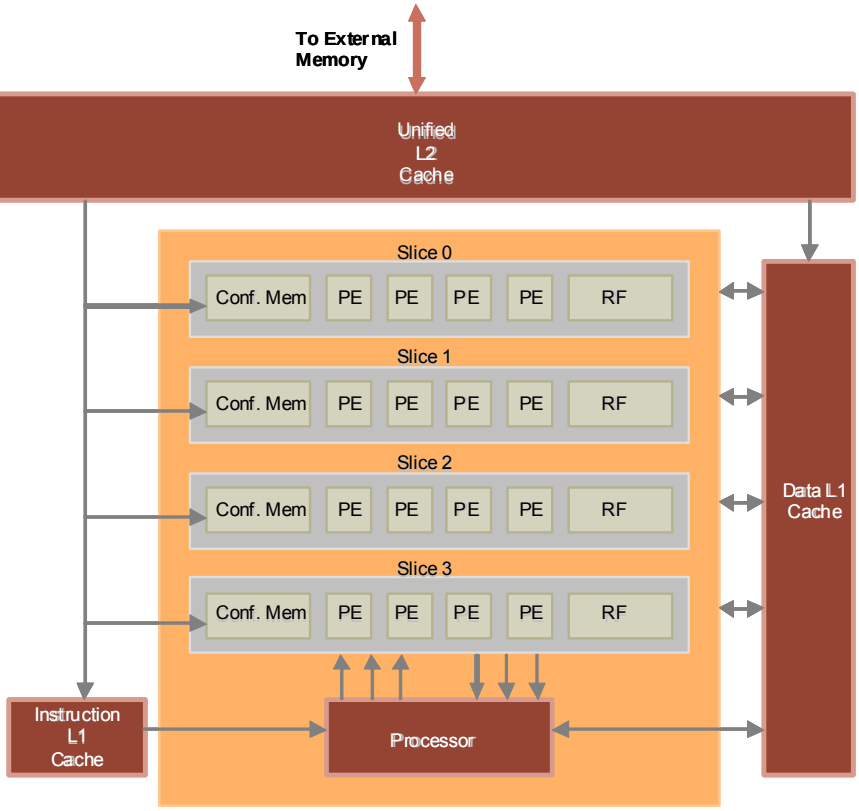


Figure 18. CRISP architecture. (RF: Reconfigurable Functional Unit, PE: Processing Element).

Each slice contains several coarse-grained processing elements (PEs), a register file, interconnection resources, and a small configuration memory. The PEs are heterogeneous HW blocks like ALUs, shifters, multipliers or memory units. Since the operations in multimedia applications are normally word-oriented these complex PEs provides better results (both in performance and energy/power consumption) than the traditional logic blocks based on LUTs and used in fine-grain reconfigurable architectures. Hence, CRISP architectures can operate at higher frequencies with lower power consumption than traditional FPGAs.

The PEs are connected together through a full crossbar, not depicted on Figure 18. This crossbar can connect the output of any PE to the input of any other PE. It is

also possible to connect a PE to a register from the main register file through the RFU ports. Each PE has a register in its output that can be optionally bypassed, just like in traditional FPGAs. By combining this optional register and the crossbar, it is possible to perform spatial computation. Spatial computation [DeHo00] allows outputs from one processing element of the reconfigurable array to be directly connected to the inputs of another processing element. Typically, only temporal computation is used in standard processors (i.e. the output from a PE is always stored in an intermediate register). Temporal computation limits the performance of the processor by forcing all elements to use the same base clock cycle while spatial computation allows a better division of the time budget. Software pipelining has also been enhanced by adding support for spatial computation. Spatial computation allows for a reduction in the number of execution cycles for loops with and without loop carried dependences, with the latter gaining the most benefit.

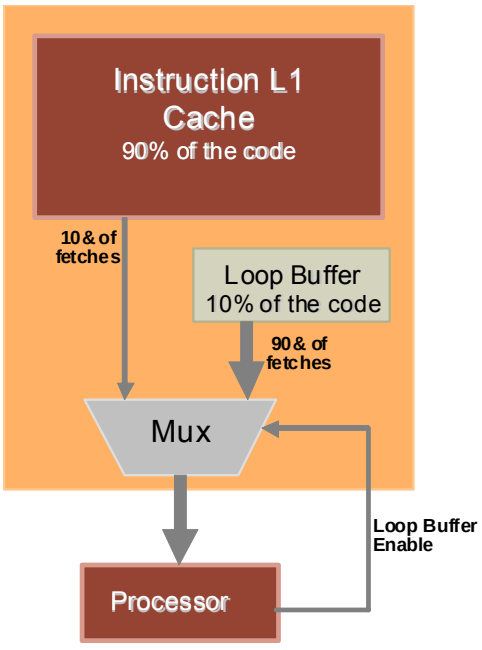


Figure 19. Loop Buffer Architecture.

In CRISP each processing element has a register at its output that can be optionally bypassed, just like in traditional FPGAs. By combining this optional register and the crossbar, it is possible to perform spatial computation. Elements in a data flow chain are connected together through the crossbar. The processing element at the end of the chain is registered to combine temporal and spatial computation.

In order to reduce the energy consumption in the instruction memory hierarchy CRISP architectures have been augmented with loop buffers (or L0 buffers) located between the instruction cache and a reconfigurable PE. Loop buffers store frequently-executed code (instructions set) in order to reduce the accesses to the more power-consuming instruction cache. This approach leads to significant energy-consumption reductions. The integration of the loop buffer in CRISP architectures and the interconnections with the other CRISP elements are depicted in Figure 19.

Each reconfigurable slice also contains a configuration memory that stores the set of instructions that will be executed in the slice datapath-components. This set of instructions can be seen as the configuration of the reconfigurable slice. Since executing a loop requires alternating quickly several configurations, the configuration memory must be able to store several configurations. As long as a configuration is stored in the configuration memory it can be loaded in just one clock cycle. In order to improve the energy efficiency of the instruction/configuration memory hierarchy the configurations are stored in small energy-efficient memories distributed among the RFUs. Loading new configurations in these memories consumes time and power. Hence it is important to minimize this overhead.

At present important research efforts are being accomplished in order to reduce the energy consumption of application running in CRISP by taking advantage of the main CRISP features. These efforts are mainly oriented to the development of techniques at compiler level, which in some cases also entails architectural modifications.

One of these techniques is presented in [RLMC06]. This technique manages to reduce energy and improve performance at the instruction memory level using the loop buffers to allow the execution of multiple loops in parallel. To accomplish this aim CRISP architecture was enhanced to support multi-threaded operations, since originally CRISP architectures only included a loop controller and therefore it was not suitable for multi-threaded operations. The developed compiler technique to manage loop-buffers and the proposed distributed instruction memory organization requires minimal performance overhead and hardware cost. The code generation is similar to the code generated for a conventional CRISP processor, except for the parts of the code that are selected for multi-threaded execution where additional instructions are inserted to initiate the multi-threaded operations. Besides, synchronization between clusters is achieved by adding an extra bit to the instructions. This instruction level synchronization is energy-efficient because it reduces the number of accesses to the instruction memory.

In [BJOD02] a software pipelining approach is applied as an effective technique for code generation for CRISP architectures. This technique, based on adding an operation assignment phase to software pipelining, performs reconfigurable instruction generation and instruction scheduling on a combined algorithm dividing

the code between the RFU and the normal function units. Although typical compilers for reconfigurable processors perform these steps separately, the results show that better results are obtained when both techniques are combined. The assignment algorithm supports the mapping of unconnected graphs onto the RFU that it is a feature normally not supported. Besides, it enables to map two or more non-connected graphs at the same time, which thereby improves the execution performance. The technique presented also exploit spatial computation inside the RFU, connecting the output of a processing element directly to the input of another processing element without the need of an intermediate register.

In [VJBD04] and [JBVC05] an algorithm is presented to explore what is the optimal CRISP loop buffer configuration and the optimal way to use this configuration for an application or a set of applications, in order to increase the energy efficiency. This loop buffer exploration methodology is based on detailed analytical energy models for software controlled clustered loop buffers. The algorithm manages to find a good loop buffer configuration and an optimal mapping for an application on the loop buffers. Different loop buffers can be evaluated to provide different area and energy-consumption trade-offs. The authors have proved their approach for multimedia applications, where loop buffering is an efficient mechanism to reduce the power in the instruction memory of embedded processors, achieving an average of 18% reduction in energy consumption, with peaks of 45% for some applications.

In this thesis we have used CRISP to generate coarse-grain architectures to test our modules. However, our approach is general enough to be applied on different coarse-grain and fine-grain dynamic reconfigurable architectures.

2.3.3. MorphoSys

The MorphoSys architecture targets data and computation-intensive applications with inherent parallelism and high throughput requirements. MorphoSys is a system-on-chip platform that integrates a general-purpose processor with RFUs (Reconfigurable Functional Units) and provides a high bandwidth data interface [Sing00].

The MorphoSys architecture is depicted in Figure 20.

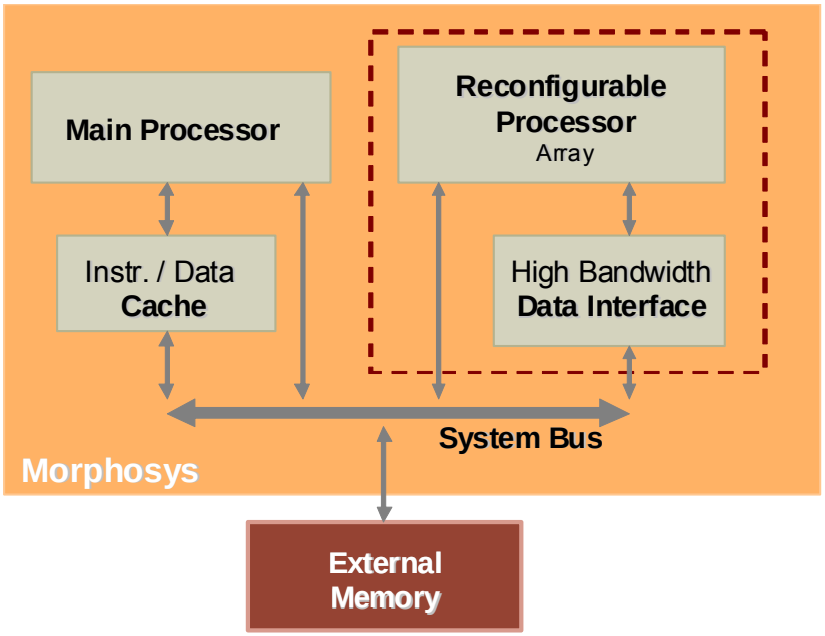


Figure 20. MorphoSys integrated architecture model.

MorphoSys is an application domain-specific system. The design of MorphoSys components is based on the multimedia and digital signal processing domain needs. MorphoSys architecture is characterized by generality and regularity. Generality means that the architecture includes more than the required number and types of processing units and interconnections for implementing a class of applications in

order to suit future needs. Regularity means that these resources are organized in regular structures. The fact that motivates this is that the target multimedia applications frequently exhibit this regularity, carrying out the same operations with a large set of data.

It is composed by three major components, namely, the reconfigurable processor array, the general-purpose processor and the high bandwidth data interface. In addition, an instruction/data cache exists for the main processor, and a system bus that interfaces with the external memory.

The communications among MorphoSys components are carried out using the system bus. In addition, the reconfigurable array has a direct high-bandwidth bus to the data interface unit.

The MorphoSys architecture takes advantage of the application parallelism using an array of reconfigurable units that can provide high performance. The high bandwidth data interface is used to satisfy the high throughput requirements of the applications.

Figure 21 illustrates the MorphoSys implementation at a block level. The Reconfigurable Cell Array (RC Array) and the Context Memory constitute the reconfigurable processing block described in Figure 20.

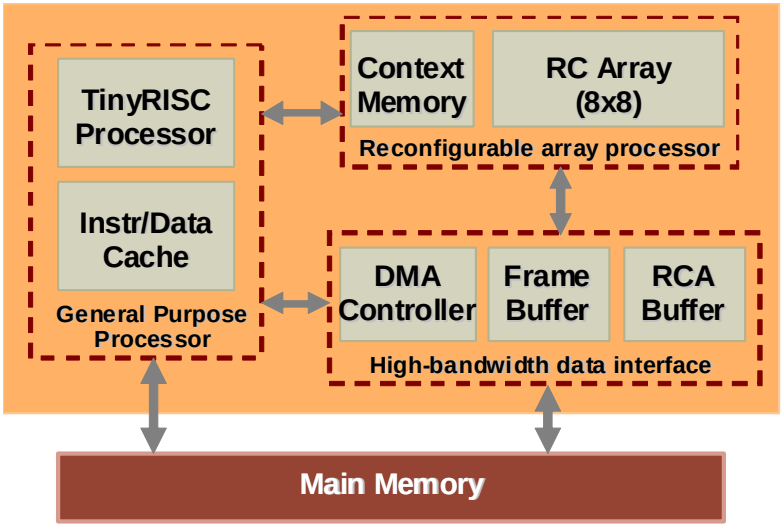


Figure 21. MorphoSys Block diagram.

The RC Array consists of 64 Reconfigurable Cells (RCs) organized in an 8 × 8 array. The RC Array has a three-level reconfigurable interconnection network, the first one for adjacent RCs, the second one for RCs in the same region, and the third one for communications of RCs on different regions.

The Reconfigurable Cell is the basic PE of MorphoSys. The configuration for the RC Array is provided by the Context Memory. The RC architecture is similar to the data-path of a conventional microprocessor, with a 32-bit operand size.

The Context Memory stores the context words used for configuring the functionality of the RCs as well as the interconnection network. The Context Memory provides one context word per cycle to the Context Register of each RC. The context word is broadcasted from the Context Memory to the RC Array along the rows or columns. Therefore, the RC Array operates in SIMD style [Nara93], with a set (row or column) of RCs receiving the same instruction (context word) but operating on different data.

The Context Memory is organized in two blocks, the column block and the row block. The column block stores the context words for column-wise operation of the RC Array. Similarly, the row block stores the context words for row-wise operation of the RC Array. A RC Array configuration plane is a set of context words that configure the entire RC Array. It comprises eight context words (one from each row or column), and its size is 256 bits.

The Context Memory can store several configuration planes. While some of the context words are being used to configure the RC Array execution, the inactive context words can be replaced by new context words without interrupting the system operation. Hence it is possible to hide the reconfiguration latency overlapping the load of a configuration with the execution of other configurations.

The general-purpose processor is a simple RISC processor, called TinyRISC, which includes a unified cache for data and instructions. The TinyRISC processor is a 32-bit processor with four pipeline stages based on the design presented in [ACGL92], which handles the sequential parts of the applications and controls the system operation. The on-chip instruction/data cache is linked to an off-chip main memory. The TinyRISC controls the execution of the RC Array, and provides control signals to the Context Memory and the data interface.

The third major component of MorphoSys is the data interface. It consists of a DMA controller and two data buffers, namely, the Frame Buffer (FB) and the RC Addressable Buffer. These data buffers store the application data for the RC Array, providing the input data and storing the outputs.

The FB is divided in two sets in order to support the overlap of computations with data transfers. At any given time, one set of the FB provides data to the RC Array and also receives processed data from the RC Array. Concurrently, the other set stores previously processed data into the off-chip main memory through the DMA controller and loads new data from the main memory for the next round of computations. This scheme hides the latency of data I/O operations that otherwise will reduce the system performance.

The RC Addressable Buffer (RCAB) is a data buffer directly addressable by the elements of the RC Array and allows to access non-consecutive memory addresses.

Along with the data buffers, the MorphoSys data interface includes a DMA controller that manages data transfers between the off-chip main memory and the MorphoSys chip. The TinyRISC processor controls the DMA providing information such as the type of operation (load/store), starting addresses in the main memory, the Frame Buffer, the RC Addressable Buffer, or the Context Memory and the number of bytes to be transferred.

The DMA controller is in charge of loading the configuration program from the main memory to the Context Memory. This is carried out before the execution of the RC Array, or sometimes even during its execution. The DMA controller is also in charge of loading application data from main memory to the data buffers, i.e. Frame Buffer and RC Addressable Buffer.

For executing applications on MorphoSys, the application may be partitioned into sequential and parallel parts. The TinyRISC is in charge of executing sequential

parts, i.e. general-purpose operations without parallelism, while data-parallel parts of the application are assigned to the RC Array.

Before starting the execution of an application MorphoSys must collect the TinyRISC instruction program, the RC Array configuration program and the application data. The TinyRISC instruction program controls the RC Array execution and also performs sequential operations on the TinyRISC. The RC Array configuration program is the set of context words representing the RC Array computations required for a given application. The application data refers to the data that the application must process (images frames, for example).

The TinyRISC instruction sequence follows a specific pattern to enable the overlapping of data and context transfers with RC Array computation. This is essential for maximizing application throughput.

Specific scheduling strategies have been included in the MorphoSys compilation flow in order to optimize its execution. One of these techniques is presented in [SFHB05]. Here the authors proposed a strategy to explore several optimizations regarding context and data transfers that can be carried out at compilation time. Two of the most important techniques in this framework are task scheduling and context scheduling heuristics. The task scheduler finds a solution that optimizes the performance through the exploration of a pruned search space. The guided search algorithm generates good solutions in the first iterations in order to reduce the exploration time. The problem of context loading management is divided into two tasks: context selection and allocation. They propose a mathematical model for context selection and two different ways to solve it. The first one is a heuristics for

context selection, which provides a nearly optimal solution, regarding performance, in just a few iterations. The second alternative is used to validate the heuristics through the exhaustive exploration of the feasible solutions. Finally, they have solved the context allocation through the exploration of a block dependency graph, so that the memory fragmentation is minimized.

In [SFHB05] a new technique to improve the efficiency of data scheduling for multi-context reconfigurable architectures based on MorphoSys model, targeting multimedia and DSP applications, is presented. The main purpose of this approach is to improve applications energy consumption. To achieve this goal, they developed a data scheduler that attempts to optimally exploit the two levels of on-chip data storage in the MorphoSys architecture (FB and the small internal RAM that each RC has (RC-RAMs)), by deciding in which memory the data have to be stored. The most frequently accessed data are stored in the on-chip memories (FB or RC-RAMs) to minimize data and context transfers, reducing the energy budget. The data scheduler also allows data and results reuse within a cluster, minimizing the memory space required by cluster execution. The experimental results of this technique have demonstrated their effectiveness reducing the execution time and power budget compared to previous data schedulers.

Finally, [RSHB08] presents a scheduling methodology for conditional execution of kernels onto MorphoSys. Conditional execution is frequently found in complex multimedia applications. In these applications the execution of some kernels depends on runtime conditions. Typically the next kernel to be processed is known after testing a condition at run-time. Then, its configurations and input data must be loaded, and

these transfers frequently produce computation stalls. In this paper the authors propose a compilation-time kernel scheduling in order to handle conditional branches by determining a kernel sequence that minimizes these computation stalls reducing the application latency. Target applications are firstly partitioned taking into account the presence of conditional branches, and then kernels are ordered for execution and mapped onto the reconfigurable system. This technique carries out an exploration of the design space of possible task sequences, obtaining good results reducing the overall execution time, which is an important factor in interactive applications subjected to real-time constrains.

However, these three techniques only provide goods results for applications with a reduced dynamic behaviour, because they are applied at compile-time. They have proposed some architectural optimizations to support highly dynamic applications and they have evaluated the benefits of applying a hybrid run-time design-time scheduling approach for dynamic applications using as a case-study a rendering application based on the ray-tracing algorithm [RiSB07].

Chapter 3:
Related Work

Run-time reconfigurable resources present many of the features demanded by next generation embedded HW/SW systems, such as high performance, flexibility and reusability. However, carrying out run-time reconfigurations often involves a costly overhead both in execution time and energy consumption. Moreover, current dynamic applications may demand reconfigurations very often significantly degrading their performance and increasing their energy consumption. In this chapter we will review different techniques developed to reduce these overheads, demonstrating that, with the proper support, the reconfiguration overheads can be drastically reduced, and dynamically reconfigurable HW can play an important role to tackle the dynamism of current applications. These techniques have been classified in three

categories: reducing the size of the configuration bit-stream, scheduling techniques and reusing techniques.

3.1. Reducing the Size of the Configuration Bit-Stream

Configurations are frequently stored in an off-chip memory. In order to execute a task the proper configuration must be loaded. This involves transferring the configuration bits from the external memories to the reconfigurable HW. As a consequence, the time needed to carry out a reconfiguration is proportional to the configuration size. So, one straightforward way to reduce the reconfiguration latency is reducing the size of the configuration bit-stream. This can be achieved by designing architectures that demand less configuration bits (coarse-grain architectures), by applying compression techniques, or by reducing the granularity of the reconfiguration allowing to reconfigure just part of the system (partial reconfiguration).

Coarse-grain architectures and partial reconfiguration have already been described in the previous chapter. Hence this section is focused on the compression techniques. These techniques apply compression algorithms to reduce the size of the configuration bit streams reducing the storage space and the data-transfer latency. Compression techniques are orthogonal to most techniques developed to reduce the reconfiguration overhead, and in fact they can even improve the results of other techniques. For instance, if a system applies a prefetch technique to hide the

reconfiguration latency, it will be easier to achieve this goal if the reconfiguration latency becomes smaller due to a compression technique.

The main drawback associated with configuration compression techniques is that the reconfigurable device must decompress the configurations online before start loading them in the reconfigurable resources. Hence, the decompress process will introduce both performance and energy overheads. In order to reduce these overheads, some authors propose adding specific HW support to efficiently decompress configurations. With this HW a portion of the configuration is decompressed while the previous one is being written in the reconfigurable device and the following one is being received. This pipelined approach can be used to hide the decompress latency. No studies exist yet about how this technique affects to the reconfiguration energy overhead, i.e., we do not know if it can also lead to energy savings, or instead of this it can incur an energy penalty since the decompress process consumes more energy than the energy saved due to the data transfer reductions.

Several approaches have been proposed in order to compress configurations. For example, Z. Li and S. Hauck presented in [LiHa01] a compression approach that takes into account the specific features of the configuration bitstream. This approach targets Xilinx Virtex series FPGAs [Xili00] and it significantly reduces the amount of data needed to transfer during the configuration process, while demanding very small modifications on the FPGA hardware. Their compression algorithm allows certain configurations to be reused as a dictionary, exploiting the regularities within the configuration bitstream. Their technique takes advantage of both inter-configuration

regularity (regularity among different configurations) and intra- configuration regularities (similar structures among FPGA rows, where configurations are grouped). In the configuration stream, some of the configurations are very similar, and by configuring them consecutively, higher compression ratios can be achieved. This approach includes a classic LZSS compression algorithm, a readback technique to build the configuration dictionary, and a configuration reordering technique that uses in a better way the regularities by reordering the sequence of the configuration word. The simulation results obtained for this approach show a reduction around to 75% of the original configuration size.

Another interesting approach is presented by J. H. Pan, T. Mitra and W.F. Wong in [PaMW04]. Just as the previous one, it exploits both inter-configuration and intra-configuration regularities. This technique takes advantage of partial reconfigurability supported by Xilinx Virtex FPGAs families. As we have mentioned before an analysis of configuration bitstreams reveals a high degree of regularity among the frames configuring the CLB array in reconfigurable devices. Besides, in Xilinx Virtex FPGAs each CLB column is configured by a set of configuration words, and words configuring common structures among different CLBs may share high regularity. Hence, among such configuration words, one configuration may be converted into another simply by flipping a few bits. This approach shows really good results compared with classical dictionary-based methods, especially when the length of the matches is small. To improve the compression rate the authors apply different schemes to generate a suitable configuration-word sequence such that similar configurations are close to each other. The best results are obtained applying a

readback scheme. This readback scheme differs from the one proposed by Li and Hauck in [LiHa01] because it uses the existing readback infrastructure in Xilinx Virtex FPGAs to read configuration bitstreams from the configuration memory to the frame data output register, carrying out the readback before the configurations are loaded in the FPGA. This approach achieves a configuration size reduction between 27 and 76% better than the previously proposed techniques in [LiHa01].

In [Kenn03], Kennedy presents a compression technique, named Banded Technique, that takes into account the regularities of the FPGA configurations. This technique is based on the fact that changed bits tend to be clustered into bands within the column because the changed bits for every configuration in the column are concentrated around the rows that are being changed. The banded technique expresses only the banded region of each configuration. This reduces the space that must be covered by the addressing scheme of the change dataset. The banded technique implementation in terms of logic and memory resources is significantly simpler and it is likely to require little silicon area despite its significant storage requirements. Kennedy also proposes a technique, named Overlay Technique, based on the fact that many of a fair portion of the bits that require a change could be set in advance without affecting the operation of the existing circuit leveraging the redundancy in an FPGA to speed up partial reconfiguration. The Overlay Technique identifies that 10% of the bits that change from one configuration to the following one can be overlaid in advance. The Banded Technique coupled with the Overlay Technique achieves an average 80% compression rate for current Virtex devices. Hence, these techniques achieve better results than full bitstream compression

techniques, and bring the advantage of demanding less HW for the decompression process. In addition these techniques are highly parallelisable.

3.2. Scheduling Techniques

As it was explained in the motivational example, the reconfiguration latency can frequently be hidden by applying scheduling techniques.

In [MKBS99], Maestre et al. presented a design-time technique for allocating data transfers and configurations to minimize overall execution time. This technique was developed for the MorphoSys coarse-grain platform and it was one of the first works that took into account the reconfigurations overhead during the task scheduling process. However, since this technique is applied at design-time it is only suitable for static applications. In [SDLB03], the authors extended this work proposing architectural improvements to support the execution of dynamic applications. Although they still do not have a complete dynamic scheduling environment, they have already evaluated the benefits of applying a hybrid run-time design-time scheduling approach for dynamic applications.

Other interesting design-time scheduling heuristics that work with task graph attempting to hide the reconfiguration latency are presented by L. Shang and N.K. Jha in [ShJh02] and by K. M. Gajjala Purna and D. Bhatia in [GaBh99].

In [ShJh02] a multi-objective hardware-software co-synthesis system for real-time, low-power distributed embedded systems is presented. For assignment they

use an evolutionary algorithm. This method can simultaneously optimized multiobjective system requirements, such as price and power. They also propose a two-dimensional, multi-rate cyclic scheduling algorithm, which determines tasks priorities based on real-time constraints and reconfiguration overhead information, and then schedules tasks based on the resource utilization and reconfiguration condition in both space and time. The FPGA scheduler is integrated in a list-based system scheduler. In their experiments the authors achieve an average schedule execution time reduction of 41.0% and an average reconfiguration power minimization of 46.0%.

In [GaBh99] an algorithm for temporal partitioning and scheduling data flow graphs for reconfigurable HW is presented. They apply two algorithms for partitioning and mapping data-flow graphs to the reconfigurable HW, namely the Level-Based Partitioning algorithm and the Clustering-Based Partitioning algorithm. The first algorithm tries to achieve the maximum possible parallelism in order to decrease the delay. The second algorithm tries to minimize the communication overhead by sacrificing the parallelism, i.e., increasing the delay. They try to extract the maximum performance from the available resources. The authors study the trade-off between maximising the computation parallelism and the communication overheads associated with such parallelism for being able to implement very large applications on small reconfigurable hardware. The potential gains of this work are in the hardware cost and speed. This approach provides good applicability for efficient reuse of the hardware and thus scalability of the available resources.

Prefetch techniques, similar to those used to hide the memory latency, have also been applied to hide the reconfiguration latency. The work presented by Li and Hauck [LiHa02] is one of the most relevant related with this topic. They propose a configuration prefetch technique developed to be applied at run-time, but based on design-time analysis. Applications are represented as task graphs but including conditions that will steer the execution. Basically conditions may trigger the execution of a task. The probability of executing a given task due to a certain condition is computed at design-time carrying out several simulations with representative input data. At run-time the task with more possibilities of being executed is prefetched. If the prediction is a success, the reconfiguration manager can hide at least part of the reconfiguration latency. Otherwise, a wrong task is loaded incurring in time and energy penalties.

Another prefetch scheduling approach to hide the reconfiguration latency is presented in [NoBa04], and [NoBa04a]. In this case the authors propose to include a HW micro-architecture that carries out the dynamic tasks scheduling for several reconfigurable units. This module manages the load of tasks into the reconfigurable resources, and tries to load in advance as many tasks as possible looking for reducing the application execution time, decreasing the reconfiguration latency overhead and increasing the resources utilization. They target a heterogeneous architecture that is very similar to the architecture that we target in this thesis: a general-purpose processor, an array of reconfigurable blocks, and shared memory resources. However, they assume that their architecture supports multiple

reconfigurations running concurrently. We do not share that assumption, since current reconfigurable devices only support one reconfiguration at a time.

As has been mentioned in [DLRJ00] the Operating System consumes a high percentage of the energy consumed in embedded systems. One way to reduce this energy consumption consists on migrate part of the operating system functionality to HW as Noguera and Badía [NoBa04b] have done with their prefetch-technique implementation. In addition, they present a whole codesign methodology and their own reconfigurable architecture. Their methodology is divided into three stages: application stage, static stage and dynamic stage. The application stage is focused on the system specifications and assumes, as we do, that the input application is specified as a task-graph. The static phase applies task transformation techniques in order to increase the architecture performance. It also implements the HW/SW synthesis and partitioning. And finally, it computes the tasks priority, assigning statically a priority of execution to each task. The dynamic stage includes the execution scheduling and the multicontext scheduling. This scheduler is implemented in HW and maintains two lists of tasks waiting for being executed. The first one contains those tasks that are ready to start their execution, (i.e. all their predecessors have been already executed). The other one holds those tasks whose predecessors are being executed at this moment, and probably will start their execution soon. The tasks in the first list have higher priority to be loaded than those in the second list. Each list is sorted by the weight of each task, which represents how critical, according to the task graph, is the execution of each task. The task with the higher weight has the higher priority for being loaded, regarding to the rest of tasks into the same list.

This technique, as we also do, uses the weights calculated at design time to choose which task, of the readies ones, will be loaded in each moment. We share the weight definition based on how critical each task is (tasks on the critical path have higher priority). The main difference between our approach and the one developed by Noguera and Badía is that our technique analyzes the whole graph, taking into account how the scheduling of each load interacts with the whole graph execution. Noguera and Badía´s technique only takes into account the current system situation to make their decisions. Besides, in this approach the load of a task can be scheduled only when all his predecessors have started their execution. Thus, it only has into account the present situation of the system and it can not take advantages of the future. Nevertheless, it was the first task scheduler implemented in HW capable of applying prefetch techniques to hide the reconfiguration latency. Since this scheduler has been implemented in HW, it can make on-line decisions very fast (just a few clock cycles) introducing almost no time overhead.

Qu et al. present in [QuSN06] another prefetch-based approach. They target a platform with multiple configuration controllers although current devices only include one. Hence they assume that different parts of the dynamically reconfigurable HW can be reconfigured in parallel. Qu also targets an architecture very similar to the one that we have presented in the introduction that includes multiple homogeneous tiles that have their own configuration SRAM that can be individually accessed. Thus, the configuration SRAM is divided into several individual sections, and controllers can reconfigure different sections in parallel, being able to load tasks in parallel improving the performance of application execution. Qu´s prefetching technique loads tasks

whenever there are tiles and configuration controllers available. Each task has a priority, and the task with the highest priority is scheduled first upon free resources are available. The priority function has into account the execution urgency of each task, how much benefit a task can get if its configuration immediately starts, and the number of successors of the task. Configurations are scheduled taking into account how many pair of tiles and configuration controllers are available at each moment. The experimental results of this approach reveal that in the test cases using multiple tiles with a single controller can speedup the system by 1.84 in average, but with multiple controllers additional 40% speedup can be achieved. However, this is a fully design-time approach, and can only be applied when tasks possess very limited dynamic behaviour.

Vikram et al. also have presented a scheduling technique that attempt to reduce the reconfiguration overhead [ViVa06], but this technique only focus on exploiting the data parallelism. In this paper the authors explain how to balance the task load of a reconfigurable multiprocessor system taking into account the reconfiguration overhead and the bus bandwidth, targeting Xilinx FPGAs [Xili05]. This work assumes that all the reconfigurable processors are executing the same task with different data. Their aim is to obtain the maximum possible speedup, for a given reconfiguration time, bus speed, and computation speed. In their approach the optimum number of Processing Units (PUs) and analytical expressions for the corresponding optimum load fractions, load transfer schedule, and processing time are calculated to get the best possible performance. They consider two different cases taking into account whether the data-transfers to the PUs can be carried out in

parallel with the PUs configurations and computations or must be carried out sequentially.

In [FuCo05], Fu and Compton present an execution environment to integrate reconfigurable HW in a simultaneous multithreaded system. This work assumes that different HW and SW versions exist of each computing intensive task. When one of these tasks must be executed, the Operating System (OS) selects the HW version if the corresponding configuration was previously loaded and is ready for execution. Otherwise, it selects the SW version in order to prevent the delays due to the reconfiguration latency. Periodically the OS decides which tasks must be loaded taking into account the execution frequency on a given period of time. This decision is made at run-time based on the status of the system; hence the OS carries out a dynamic task scheduling. The implemented scheduling algorithms not only choose which kernels should be implemented in hardware for each scheduling interval, but also the specific hardware implementations for those kernels. Multiple hardware implementations of a kernel may be available in order to allow for a varying trade-off between speed and area. This process is used to balance the hardware resources among the competing threads, and choose the best combination of hardware for each scheduling interval. They first present two simple and fast-executing heuristics based on a greedy method. And finally they proposed a scheduler that uses the Multi-Constraint Knapsack Problem (MCKP) [MaTo09] to model hardware scheduling. Their results show that this last multiple kernel scheduler allows for greater overall acceleration, more than 20% over software-only execution.

3.3. Reuse Techniques

When the same tasks must be executed recurrently in the reconfigurable HW a straightforward way to reduce the reconfiguration overhead is to reduce the number of reconfigurations needed by reusing the configurations that were previously loaded.

One way to maximize the reuse of configurations is to consider that a dynamically reconfigurable HW device is similar to a cache memory where configurations are stored. When a new configuration has to be loaded, it is necessary to find enough free space. If this is not possible, one or more configurations are replaced by the new one. The key issue is deciding which configuration should be replaced among all the loaded configurations in order to maximize the possibilities of reuse.

Li et al. [LiCH00] developed different configuration caching techniques for five different dynamically reconfigurable HW models, namely single context model, partial run-time reconfigurable model, multi-context model, relocation model, and relocation/defragmentation model. They have developed both lower bound, to quantify the maximum achievable reconfiguration reductions possible, and realistic caching algorithms for these structures that can significantly reduce the reconfiguration latencies. The simulation results of this approach demonstrate that the partial run-time reconfigurable architectures and the multi-context architecture are significantly better caching models than the traditional single context model.

For these architectures they propose several reuse algorithms that can be classified into three categories: run time algorithms, complete prediction algorithms,

and general off-line algorithms. The run time algorithms use only basic information on the execution of the program up to that point, and thus must make guesses as to the future behaviour of the program. LRU (Least recently used), is one of these algorithms. Because of the limited information at run time, a prediction of keeping a configuration or replacing a configuration may not be correct and can even cause higher reconfiguration overhead. The complete prediction algorithms use entire execution information of the application. These algorithms attempt to search the whole execution stream in order to lower the configuration overhead. They provide the optimal (lower bound) or near optimal solution, but they entail high computational expenses. Finally, the general off-line algorithms use profile information of each application, with computationally tractable algorithms. They represent the most realistic solutions for the case where static execution information is available, or approximate algorithms where highly accurate execution predictions can be developed.

Another approach that also proposes configuration replacement algorithms to optimize the configuration caching problem is introduced in [SuNG01] by Suraj et al. In this paper the authors propose a history-based online algorithm that, based on simulation results, outperforms previous cache replacement algorithms. This algorithm tries to predict the future sequence of configurations based on recent history, in order to evict the configurations that will be used in the latest place. To carry out the replacement decisions this approach maintains a chain with the configuration that last followed each configuration. The evicted configuration is determined by following this chain. The configuration on the fabric that occurs furthest

in this chain is predicted to occur furthest in future and will therefore be evicted. The strategy used by this history-based algorithm is not the same as Most Recently Used (MRU) because it introduces a control mechanism to avoid the thrashing problem that may happen when applying MRU and that can drastically increase the average reconfiguration latency by more than 160%. The replacement policy must take a decision very fast. Hence, the authors propose to implement the replacement algorithm in HW. The HW implementation of this algorithm requires storing the chaining information of each configuration. Since this can be too expensive in hardware, the authors propose to store only the configurations that are currently in the fabric.

As we have mentioned before the total reconfiguration latency depends not only on the number of times a configuration is loaded but also on its size. It is therefore possible that few loads of a very large configuration will generate more delays than many loads of a smaller configuration. Thus, it might make more sense to keep larger configurations in cache longer and to consider both size and frequency when making eviction decisions. Due to this, the authors also consider the configuration size by assigning a cost to each configuration in the fabric and whenever a configuration is accessed, its cost is set to some large constant while the cost of the other configurations on fabric are reduced by (fabric size - configuration size). This penalizes smaller configurations more than the larger configurations. Hence during replacement, a configuration that has the smallest cost has more possibilities for being evicted. Besides this approach extend the caching model to a three-level cache model taking into account the defragmentation and the exclusion

property for performance variations. This history-based algorithm shows consistently competitive performance ratios with offline algorithms, because it replaces the configuration that it estimates to occur furthest in the future. In fact, this algorithm tends to make choices similar to that of Belady's [Bela66] offline algorithm that guarantees the optimal results.

The possibilities of hiding the reconfiguration latency and reusing previously loaded configurations can be greatly improved by introducing a more complex configuration memory hierarchy. For instance Trimberger et al. [TCJW97] and Lehm et al. [LPPA02] propose multi-context FPGAs. These FPGAs can store multiple configurations and change from one to another in just one clock cycle. The idea is to add memory elements to store all the configurations and select at run-time the appropriated one using multiplexers. Unfortunately, for FPGAs this is an expensive approach, and no commercial FPGAs implement it. However, the idea is feasible for coarse-grain, and many platforms include it [SLLK00], [Hart01].

Another way to increase the reuse ratios is taking advantages of loop-unrolling techniques. M. Sánchez-Élez et al. [SFHM02] propose to use a loop-unrolling technique to reduce data and contexts loads. They target applications where a sequence of tasks is executed periodically and they evaluate how many times this loop can be unrolled, taking into account the storage requirements of each iteration and the size of their internal memory, in order to reduce the number of reconfigurations needed.

As we have mentioned before, the different techniques we propose that focus on reconfiguration overhead reduction are compatible, and they can be coupled in the

same system in order to increase the minimization of this overhead. For example, Resano et al. [RCGG07] introduce a manager for HW multitasking reconfigurable systems implemented in HW that attempts to optimize the reconfiguration process applying a reuse technique to reduce the reconfigurations thrashing, and prefetch techniques to hide as much as possible the time overhead due to the reconfigurations. Besides, this manager provides two prefetch modes, namely a greedy approach and a scheduler-based approach. The first one is easier and faster to implement than the second one, but it provides worse results as well. The experimental results show that the manager can deal with complex application schedules, generating almost no delays due to its computations, while hiding about 70% of the reconfiguration latency. The main drawback of this approach is that the HW needed to implement the manager has been optimized for performance and may become too expensive for big systems. So we address the remaining limitations in this thesis.

Chapter 4:
Base Methodology

The target architecture of this work is a multiprocessor system that includes several reconfigurable units (RUs). These units will behave in a similar way than any other processor in the system with the exception of the need to carry out run-time reconfigurations in order to load tasks.

Developing a multiprocessor reconfigurable system is not an easy problem. Fortunately, during the recent years IMEC has developed an interesting environment that solves most complex implementation issues, the ICN model ([MBVL02], [MMBM03], and [BMNM03]). Our approach is not bounded to this solution, but we have taken this model into account while developing our techniques to reduce the reconfiguration latency, since we believe that it is the most interesting approach available to develop a multiprocessor reconfigurable system on top of an FPGA.

To deal with this multiprocessor system that includes both conventional processor and reconfigurable processors, we need a scheduler capable of dealing with a highly heterogeneous platform.

Again this time, the work carried out at IMEC provides an interesting solution. The Task Concurrency Management project (TCM) [YDCV00], [YWMC01], [WMYP01], [MJSY02], [LWMV02], [YaCa03], [MMSY07]) is a management environment developed to handle task-graphs concurrency in heterogeneous multiprocessor systems. Although the techniques developed in this thesis are not limited to TCM, we have used it as a reference for our work, and we have integrated our techniques in the existing TCM environment.

This section provides some inner details about the ICN model and the TCM environment.

4.1. The Inter-Connection Network Model

Normally, loading a HW task on a given FPGA region involves carrying a complex placement process to assign the task computations to specific FPGA resources. In addition, a routing process is needed to establish the required connection between the different resources that compose the task. Typically, these two stages are carried out at design time, assuming that all the FPGA resources are available. However, this approach is not applicable to a multiprocessor reconfigurable system, where several tasks will share the FPGA, running in parallel in different

FPGA regions, since two different tasks may attempt to use the same resources at the same time.

Hence, although FPGAs allows loading new tasks at run time using partial reconfiguration, the classical design process is not compatible with this option. An easy solution will be to carry out the placement and routing processes at run-time. Unfortunately, these processes are extremely complex and time demanding and this approach will remove all the advantages of partial run-time reconfiguration, since carrying out the demanded computations may consume several minutes and this delay is unacceptable in multi-tasking systems. Hence, to take advantages of the possibilities that partial reconfiguration offers it is necessary to find a way to load tasks at run-time in affordable time. Besides, the system must be able to determine at run-time where to load the tasks in order to take advantages of the FPGA resources available at each moment. The ICN model solves this problem dividing the FPGA onto a set of identical units, or tiles, with a fix intercommunication interface, which is known at design time. Since the interface is known, designers can carry out the placement and routing processes at design time. Moreover, since all the units have the same interface, a task properly compiled can be loaded in any of them without any extra computation cost.

The basic structure of the ICN model is depicted in Figure 22.

The ICN model divides the FPGA into a set of identical reconfigurable units, where each one of these units can execute a task. Besides, every unit is wrapped by a fixed communication interface. This interface provides support to carry out inter-tasks communication using message passing primitives over an interconnection

network integrated into the FPGA. This network is also used by the conventional processors.

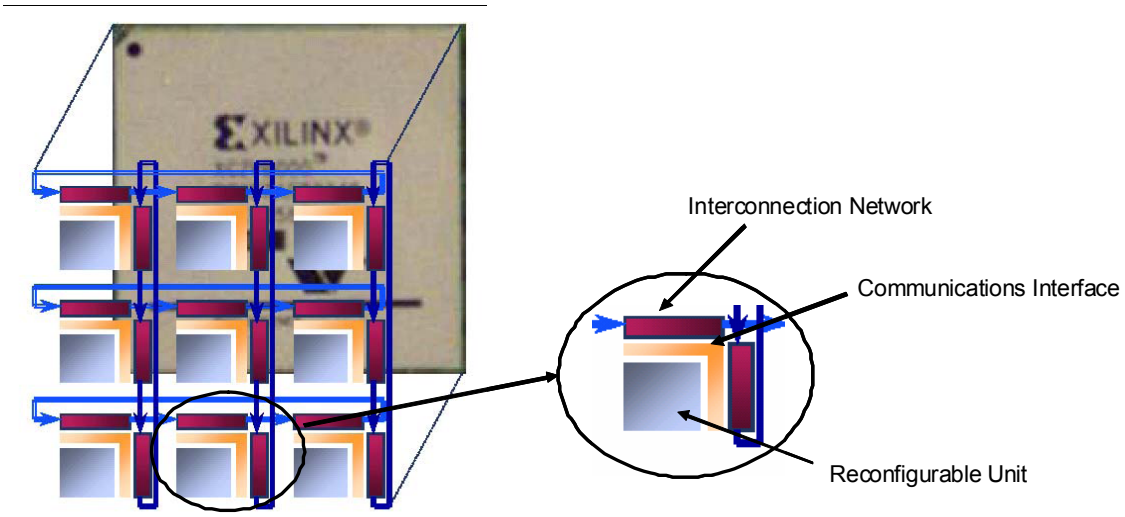


Figure 22. ICN model for FPGAs.

The communication protocol is the same for those tasks executed in HW than for those tasks executed in SW, because they use the same message passing primitives. Hence, with this approach a task scheduler can manage HW and SW resources in a similar way.

The ICN communication protocol enables the synchronization and communication between tasks on an easy way from the designer point of view. Apart from this communication protocol ICN integrates some basic OS primitives in order to provide a basic support to the designer [MNCV03], [NCVV03], [NMBV03], [MMBM03], [NMVM04]. This small OS is called OS4RS (Operating Systems for Reconfigurable Systems) and provides the following functionalities:

- Task creation and deletion at execution-time.

- Dynamic tasks assignment.
- Communication between tasks.
- Debugging support for tasks execution.
- A sniffer as a network monitor tool.
- Bandwidth-control assign to each task.

As we have mentioned before, ICN model is by nature a heterogeneous architectural model, not only because it can deal with HW or SW resources in a similar way, but also because in order to include the OS support at least one conventional processor must be included in the system.

Figure 23. Platform GECKO

The ICN model has been successfully implemented in platforms with SA-1110 and Power-PC processors; and Virtex and Virtex II FPGAs [MBVL02], [MMBM03].

One example is the platform GECKO presented in Figure 23. This platform consists of a PDA (personal digital assistant) Compaq iPAQ 3760 and a FPGA Xilinx Virtex 2. The PDA includes a processor StrongARM SA-1110 and an expansion bus allows connecting the processor to the FPGA. The communication interface is implemented inside this FPGA, coupled with the interconnection network and two reconfigurable units.

To sum up, the ICN model provides a vision of an FPGA similar to a multiprocessor system where tasks can be assigned either to reconfigurable units or to conventional processor. Besides, ICN provides the needed support to integrate the reconfigurable resources into a heterogeneous multiprocessor platform. Due to these features it is possible to use a multiprocessor scheduling environment like TCM [MMSY07], initially developed for conventional processors, to manage also a set of reconfigurable units.

4.2. TCM

Figure 24 depicts the different steps to schedule an application in this methodology.

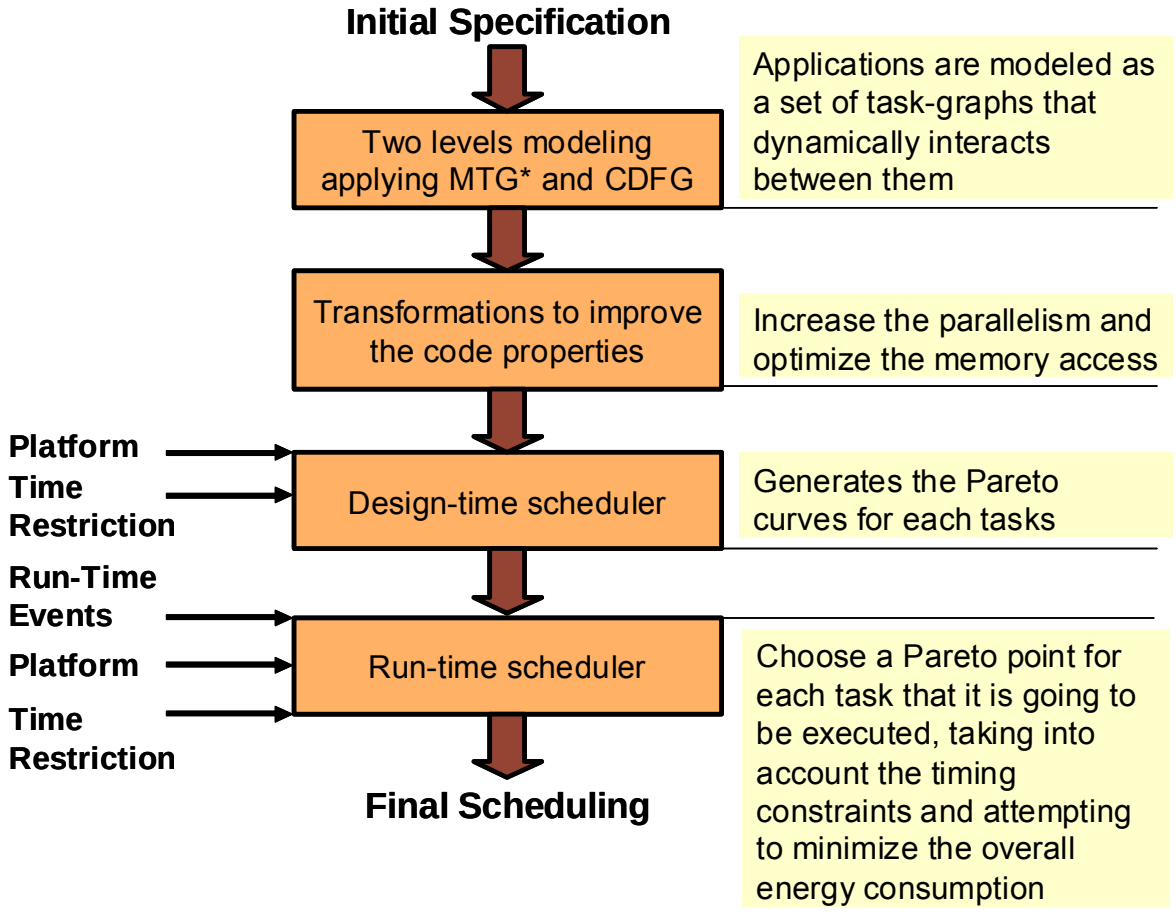


Figure 24. TCM scheduling methodology flow diagram.

TCM is based on three main ideas:

- Applications are modelled using a two-level hierarchy. In the top level an application is described as a set of task-graphs that dynamically interact among them. In this level TCM proposes to use a description model specially designed for this purpose: MTG*. In the second level, each task-graph is described as a set of tasks. These tasks are the basic scheduling unit. The idea is that these graphs of tasks should include only limited non-deterministic behaviour in order to analyze them at design-time.

In TCM the basic scheduling unit, which in our system is a task, is called thread-node, and what for us is a task-graph in TCM is named thread-frame.

- If the execution of a task-graph heavily depends on external data. TCM proposes to develop several task-graphs. Each one of these graphs represents an important execution scenario. These important scenarios are identified at design-time after profiling the tasks. At run-time the scheduler will analyze the external data and select the proper scenario.
- TCM proposes a hybrid design-time/run-time approach to carry out the task scheduling process. The idea is to carry out most computations at design-time while taking into account the run-time events and the system state in order to meet the application timing constraints while minimizing the system energy consumption.

4.3. Modelling Applications in TCM

Applications are modelled using a two-level hierarchy:

On the upper level applications are described as a set of tasks graphs interacting among them. This level is described using the MTG* model [ThCa00] that is an extension of Petri's networks [Mura89] developed to provide the needed modelling support to describe inter-tasks dynamic and non-deterministic interactions.

The lower level describes each task-graph as a set of tasks, where each task is a piece of code with enough entity to be separately assigned to a reconfigurable unit or processor. At this level non-deterministic events are not allowed, and the dynamic behaviour is limited to loops and conditions that only depend on entry data. To sum up, TCM only allows a limited dynamic behaviour of applications inside each task-graph. And therefore, the main part of the dynamic behaviour, and above all the most unpredictable behaviour, must be kept in the upper level. The aim of this division is to separate the part of the application specification that can be managed and optimized at design time, from the one that can be managed at execution time.

4.4. Task-Graph Scenarios

Due to the fact that a task-graph still includes a certain level of dynamism, the execution of task-graphs cannot always be characterized with the requested accuracy using a constant execution time. For example, if there are conditions inside the graph, the execution time varies depending on the input data. To solve this problem the most common approaches consist in characterizing each task-graph with the maximum possible execution time (worst-case), or with the average execution time. The first option frequently is a too pessimistic and unjustified scenario, because usually the worst possible case is very infrequent. The second option is probably more realistic, but using that approach may lead to timing constraint violation each time that the actual execution time is far from the average execution time. Hence, this approach is not applicable if the system wants to provide a constant quality of service level.

TCM propose to solve this issue using several scenarios for each task-graph [GPHV09]. A scenario represents the execution time of the task-graph for a given range of the input data values. Each scenario is represented with its own task-graph. Frequently, all these graphs share the same structure, but with different execution times. The number of scenarios must be chosen carefully in order to prevent an explosion of scenarios, this should be done carrying out an extensive profiling phase in order to identify the most significant scenarios. In order to reduce the number of scenarios similar scenarios can be grouped together and a worst-case approach can be used to define the execution time of each task-graph in the group. Since, this worst-case approach is only applied to merge similar scenarios, is should not fall into the disadvantage of conventional worst-case approaches.

4.5. The Design-Time Scheduling Phase

The TCM scheduling process starts at design-time analyzing the task-graph. As these graphs mainly include static behaviour, this analysis can be carried out accurately. Basically the design-time scheduler explores for each graph the different placement and scheduling possibilities. Nevertheless, due to the fact that the conditions in which the task-graph is going to be executed are only known at run time, it is not possible to foresee what is going to be the best solution, hence, the design-time scheduler does not know whether it should optimize the scheduling to provide maximum performance, or to reduce the energy consumption, or to achieve a given energy/performance trade-off. To overcome this limitation the TCM design-time

scheduler does not generate a single solution but a set of different solutions with different performance/energy-consumption trade-offs. Each one of these solutions represents a mapping of the tasks to the different processors or reconfigurable units, and the execution schedule. With these solutions the scheduler generates a Pareto curve. This kind of curves is usually used at systems with several parameters to be optimized [EsKO90]. In this curve each point represents a pseudo-optimal solution that guarantees that no other explored solution is better in the two optimization parameters that have been taken into account: performance (execution time) and energy consumption. Hence, it cannot be said that a Pareto point is better than other. Nevertheless each one is more suitable for a certain situation, and at run-time, the scheduler will select the proper Pareto point for each task. On the one hand, if at a certain point of time the workload of the system is high, the run-time scheduler may select a Pareto point with a schedule optimized for performance. On the other hand, if the workload is low or the timing constraints are not very demanding the scheduler can select a Pareto point optimized for reduced energy consumption.

4.6. The Run-Time Scheduling Phase

The design-time scheduling phase generates a Pareto curve for each scenario of each task-graph. The objective of the run-time phase is to select the proper Pareto point for each task-graph that must be executed and decide the task-execution sequence. To accomplish this objective, firstly the run-time scheduler identifies the active task-graph and selects the appropriated scenario for each one of them taking

into account the input data. Afterwards, the scheduler will attempt to select the combination of Pareto points and task-execution sequence that meet all the timing constraints while minimizing the overall energy consumption. The algorithms used to carry out the chosen of the Pareto points are described in [YaCa03] and [YaCa04].

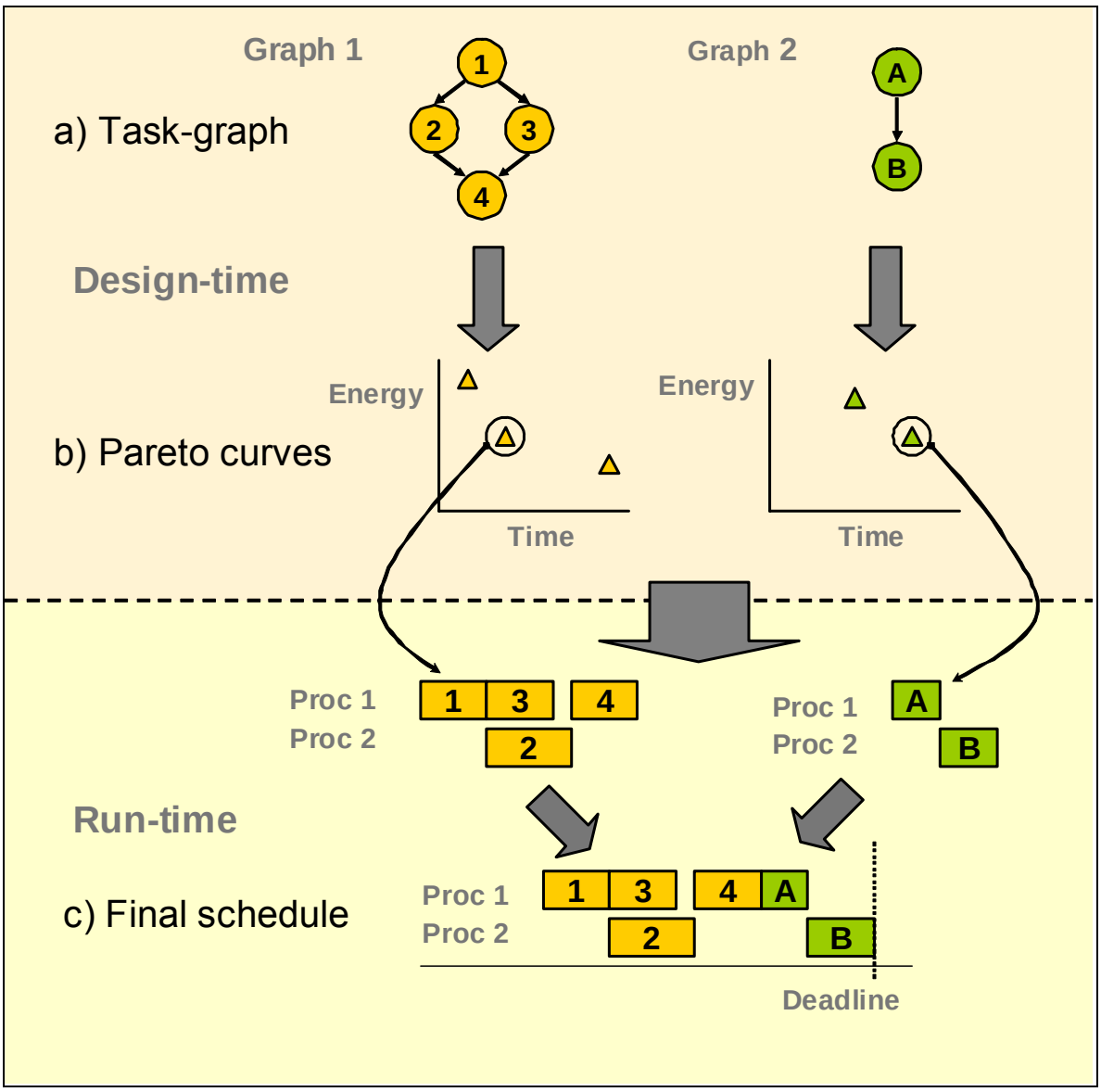


Figure 25. TCM scheduling process. The target platform has two processors with different performance and energy consumption features.

Figure 25 depicts the different stages of the scheduling process. For simplicity in this example each task-graph only has one scenario. After exploring the possible scheduling of each graph, a Pareto curve is generated for each one of them. Finally, at execution time the scheduler generates a final scheduling, choosing the Pareto point for each task-graph to be executed. In this case the scheduler has chosen for Task 2 the Pareto point that consumes less energy. Nevertheless, it has not made the same decision for Task 1, because it will not meet timing constraints. Hence, the scheduler has selected an intermediate Pareto point which consumes a little bit more of energy but meets the timing constraint.

The main contribution of this scheduling process is to provide flexibility at run-time, while carrying out the exploration of the scheduling solution space for each task-graph, which is the most time consuming task of the whole scheduling process, at design-time.

4.7. State-of-the-Art in TCM

4.7.1. Task-Graph Interleaving

As we have explained before, applications at TCM environment are composed by multiple concurrent task-graphs. And they are decomposed into tasks which are in turn assigned and scheduled on multiple different RUs. Right now the task-graphs scheduling technique carries out performance-energy tradeoffs explorations for each individual task-graph and used the exploration results at run-time to fulfil system-level

constraints. However, they did not exploit the fact that the concurrent task-graphs can be executed in an overlapped fashion.

[MaSC07] presents an on-line technique that performs task-graphs overlapping by run-time tasks re-scheduling. This technique achieves better performance results without extra energy consumption joined to the TCM task-graphs scheduler.

Before individual task-graph schedules often exhibit idle slots. This is due to limited task parallelism within a single graph, which leads to the insufficient utilization of the processing capacity of the platform, and different execution times and energy consumptions caused by running the same task on heterogeneous processors: consequently, power-optimizing design-time scheduling strategies tend to under-utilize the parallelism in favour of the energy efficiency.

Because a traditional re-scheduling of all tasks would be extremely time-consuming, the previous run-time scheduler can only run over all Pareto-optimal scheduler sequentially and thereby ignore these intra-task graph slacks. As a result, in the global schedule, these slacks are still present. In contrast, the proposed interleaving technique can efficiently reduce those slacks.

This interleaving technique consists of two steps. The first step produces an initial interleaved global schedule using the greedy strategy. Starting from this initial global schedule, the second step then does a local search based on a unique neighbourhood mapping function. The greedy behaviour of the first step is compensated by the local search carried out for the second step and thereby it optimizes the performance of the final global schedule.

At this approach the mapping of tasks to RUs and the task order are preserved. This means that each task is mapped to the same RU as in its original schedule, and that if one task is scheduled before or after another task in the design-time schedule, the same condition holds in the run-time schedule.

Interleaving only operates rigid translations of tasks along the time axis of the RU where it was allocated at design-time, in order to fill in idle slots. Hence, interleaving speeds up concurrent task-graphs without increasing the energy consumption with respect to sequential execution. The hierarchical scheduling adopted by this approach reduces the exploration space, compared to the conventional flattening-and-rescheduling (FAR) schemes such as [StRa91] and leads to lower run-time overheads.

This interleaving technique has been proved for a set of randomly-generated task-graphs and also for an algorithm from real-life video and image processing applications implemented on a dual-processor TI TMS320C6202 board. Interleaving technique has obtained improvements over non-overlapped tasks, and it also has less overall energy consumption than a previous DVS method for real-time tasks. It has achieved a reduction of 22-29% in the application execution time, while the impact of run-time scheduling overhead proved to be negligible.

4.7.2. TCM Estimation Environment

In order to take full advantage of the TCM scheduling a fast an accurate method to estimate the execution time and the energy consumption of each scenario

is needed. Traditional approaches do not give an answer to all TCM requirements at the same time, because they are too slow, do not work with the source code, cannot provide independent estimations for each task-graph, or are unable to deal with dynamism. Recently Scarpazza and Brandolese proposed [ScBr06] a fast estimation technique that works with the source-code. This technique is able to estimate both the execution time and the energy consumed by an arbitrary portion of a given source code. Besides it is able to exploit all the contextual information related to the different scenarios of each task-graph, estimating the accurate cost of a task for each one of the scenarios generated for its respective task-graph. Another advantage of this estimation approach is that it does not use the target platform but only a pre-characterization of it. Hence, it is possible to obtain estimations even when the platform is not available.

The core of [ScBr06] technique is the estimation of the time and energy cost of the task-graphs obtained from the input source code assuming the simplifying hypothesis that the execution and energy cost of a node can be expressed as the product of its single-execution and the number of times that it is executed. This approximation is valid within negligible errors because processing units typically comprise components with limited latency variability. This technique computes single-execution costs during a static analysis based on an abstract translation model [Scar06], and determines the execution counts by running an instrumented version of the original application over actual input data. Moreover, this technique externally operates in the same way than a compiler, for instance as a GCC, therefore it can be easily integrated in the TCM development environment.

The experimental results for this approach, presented in [ScBr06], [Scar06] and [SRNC06], prove that this estimation technique it is accurate and remarkably faster than instruction-set simulation. In addition it provides interesting advantages compared with traditional approaches. Since traditional approaches use an instruction-set simulator, it is an accurate solution but too computationally demanding. Other works use compilation-based estimation techniques, for example the one presented in [LaLS99] uses a control-flow graph obtained from a compiler. They also rely on compilers annotation to obtain some needed information. The problem of this approach is that is too compiler dependant, and hence cannot be use at source-level.

4.7.3. Real-Life Applications in TCM

The merging of computers, consumer and communication disciplines gives rise to very fast growing markets for personal communication, multimedia and broadband networks. Technology advances lead to platforms with enormous processing capacity that is however not matched with the required development of system design processes. As we have mentioned before, one of the most critical bottlenecks is the very dynamic concurrent behaviour of many of these new applications. They are fully specified in software oriented languages (like Java, UML, SDL, and C++) but they need to be executed in real-time cost/energy-aware way on the heterogeneous SoC platforms as the ones following the ICN model.

TCM has been illustrated on several real-life applications. The quality of service (QoS) adjustment algorithm of a 3D image rendering application has been tested

applying TCM methodology in [YLWM02]. In this work, TCM methodology is used for MPEG21 based demonstrator mapped on a multi-processor simulation platform with a hierarchical share memory organization. The 3D decoding/rendering typical by performs a 2D texture and a 3D mesh, being first decoded and then mapped together to give the illusion of a scene with 3D objects. This kind of 3D rendering requires that each frame of the rendering process is recalculated completely. The required computation power depends significantly on its number of triangles per frame. In his approach, when the available resources are not enough to render the object, instead of letting the system break down, the corresponding mesh of the object can be degraded, reducing the number of triangles per object, to decrease the resources consumption, while maintaining the maximal possible quality. Hence, for this algorithm the number of triangles that are used to describe a mesh can be scaled up or down. For a 3D object, the more triangles used to represent a mesh, the more precise the description of the object. However, it slows down the geometry and rasterizing stages because they demand more computations. Consequently it decreases the number of frames generated per second. For a given computation platform and a desired frame per second rate, the number of triangles that can handle in one frame is almost fixed. Based on the number and type of objects in the current frame the QoS controller will assign the triangles to each object so that the user can get the best visual quality for the given frame rate and the available computational resources (this experiment has been explained in [YLWM02] in detail). In the QoS kernel of the considered 3D application, for each visible object on the scene, a separate task-graph will be triggered, in which the number of triangles is adjusted to

the number specified by the QoS algorithm. Each time that a frame starts, depending on the number and type of the visible objects, the QoS controller will adjust the number of vertices assigned to each object, in order to provide the best quality at a fixed computation power. The experiments show that the TCM run-time scheduler applied to the QoS adjustment algorithm for 1000 frames can achieve energy savings around 58% for 5 frames per second rate and 40% for 10 frames per second one, while the deadline miss ratio remains the same. If the time constrains are hard to meet the TCM method will automatically select the Pareto point that provides the best performance for all. However, when they are less tight, the scheduler will select a set of Pareto points that meet the deadlines but with the minimum possible energy consumption.

Another experiment where TCM has been applied to a real-life application is the one presented in [MWHD03]. In this case, the TCM methodology has been applied to the Visual Texture Coding (VTC) decoder of the MPEG-4 standard. In this work the authors extract the basic task-graph for the VTC decoder, and then, they carry out some transformations to increase the concurrency and to handle the variable workload, introducing system-level trade-offs. The transformation step has two phases, namely, refining the task-graph, and clustering the tasks. After the transformation, the task-graphs are analysed and profiled in order to obtain the most representative run-time scenarios. In this application the number of graphs to execute for each frame is constant, but the appropriated scenario for each task-graph varies from one frame to the following one. Hence, it is a good example of a dynamic application. TCM takes advantage of the run-time variations in order to achieve

energy savings. In the experiments carried out with representative input data the TCM scheduling approach provides an energy saving of 55% compared with state-of-the art scheduling approaches.

4.7.4. Current TCM Open Questions

TCM is alive. Currently several open questions exist that are being addressed in new ongoing research activities at both IMEC and its university partners co-operating on the TCM project. Some major open questions are:

- Cost parameters quantization: Although TCM is right now focused on the trade-off space of the system performance and energy consumption; it is also applicable to other cost parameters.
- Task model extraction: Task model extraction is a crucial pre-processing stage for TCM, however it is still conducted manually and is thereby often a tedious and error prone procedure.
- Look-ahead design space pruning: the design-time scheduling algorithm could be further accelerated with more aggressive pruning techniques.
- Optimality analysis: the lower bounds of either the time-budget or the energy-cost by an analytic method can further improve the scheduling quality of the basic design-time scheduler.
- Run-time Pareto-curve calibration: at the run time scheduling stage, it is possible to have only partial knowledge of the dynamic context. Hence,

the schedule is made based on the partial knowledge and a prediction and later it is refined when the full knowledge is available.

- Synchronization points: the run-time software multi-threading technique has so far only considered the interleaving of entire thread frames, hence, right now no new task-graphs are allowed at run-time. To support it involves the problem of defining the synchronization points in a task-graph where arriving task-graph are allowed.
- Efficient real-time OS synthesis: a better integration of applications and the underlying platform is desirable.
- Local memory conflicts: The way to handle this issue relies on efficient usage of data memory hierarchy and better data access scheduling including data reuse explorations.
- Software and Hardware reliability: integrating more and more software components requires a compromise because little or no information about these components' reliabilities is available and a comprehensive test is often infeasible. Hence, systematically probe these software components at design-time/run-time and hence formulate a reliable integration becomes an interesting problem.

4.8. Example of the ICN/TCM Integration

In this section, we are going to use a simple multimedia application, a video decoder based on the JPEG standard, to illustrate how the TCM scheduling approach can be applied to reconfigurable systems that follow the ICN model for reconfigurable HW. This application receives periodically an image that is decoded in four stages, as it is depicted in Figure 26.

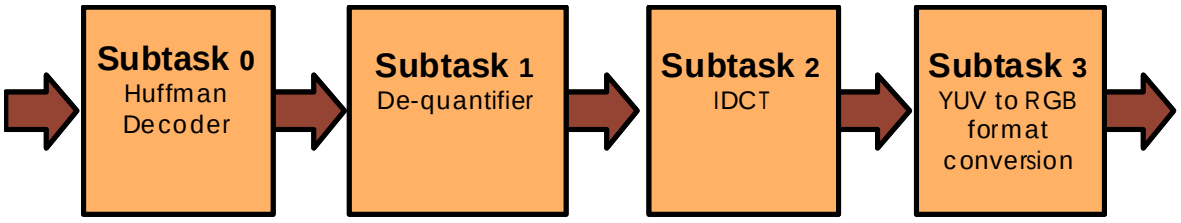


Figure 26. Motion JPEG video application task-graph.

Firstly, the entrance to the process is a description in C++ of the application. From this description, a specification in the MTG* model is generated. Here the different dynamic interaction between the task-graphs of the application are described, nevertheless in this example it is not necessary to do this due to the fact that the whole application is a single task-graph. Therefore, it is good enough for this example to build a single task-graph from the initial specification.

In the following step each task is refined using OCAPI-XL [VoMa06] that is a tool to develop equivalent HW and SW versions of a task starting from C++ code. OCAPI-XL uses a refinement process that step by step introduces in the code typical time considerations needed for the HW design (like introducing a clock signal) in such a way that once the process has ended a task description can be automatically

generated in synthesizable VHDL. In addition, starting from the same specification a C compatible version is generated for each task. C and VHDL code generation are carried out automatically once enough information has been introduced at the original code, and it is guaranteed that both versions are functionally identical, since both are generated from the same internal model. This equivalence is an indispensable feature to be able to move a task at execution time from a SW resource to a HW one or vice versa.

OCAPI-XL, apart from generating VHDL and C codes, allows simulating cycle by cycle the execution of each task not only for the HW version but also for the SW version. In order to achieve energy estimation it is necessary to turn to other tools since OCAPI-XL does not include an energy consumption model. In this case we have use XILINX Xpower [Xpower] tool for the task HW version and the ARMulator simulator [Arm05] for the SW version. Once the time and energy consumption features of each task are known, the task-graph, which will be used for TCM scheduler, is generated.

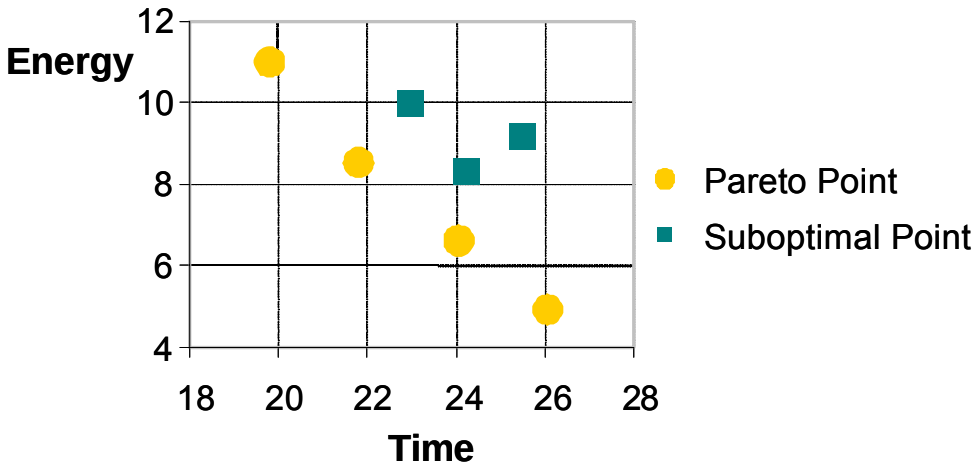


Figure 27. Pareto curve for Motion JPEG video application.

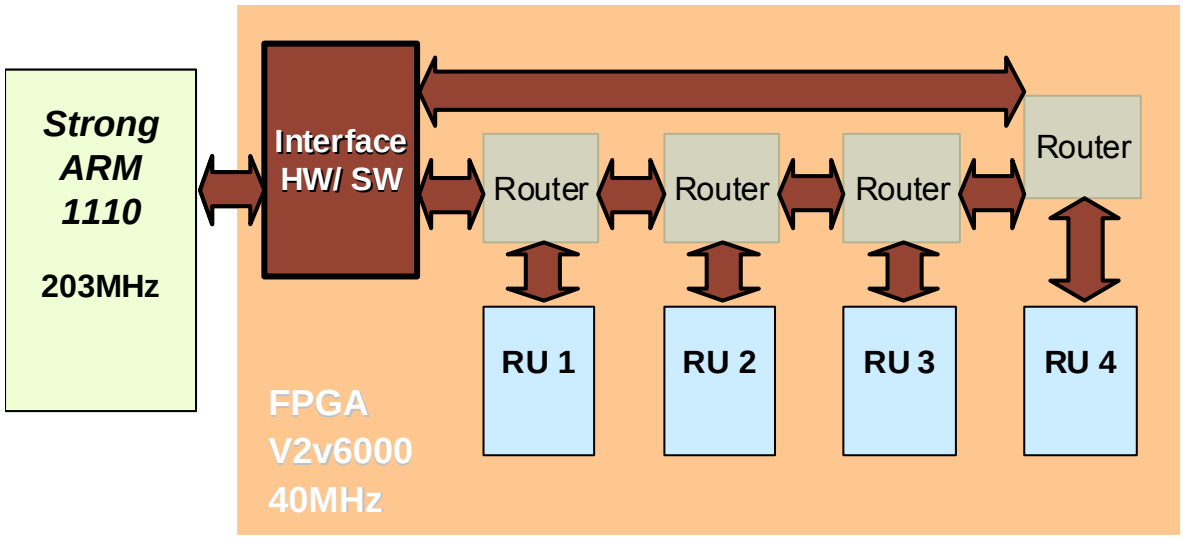


Figure 28. Target architecture for Motion JPEG video application.

During the design-time scheduling phase the scheduler will generate a Pareto curve using the graph and a description of the target architecture. This Pareto curve is depicted in Figure 27. In this case the target architecture (Figure 28) is a HW/SW platform with a StrongARM SA-1110 processor and a FPGA Virtex2 v6000 FPGA divided in four Reconfigurable Units (RUs).

Figure 27 includes not only the Pareto points, but also some suboptimal points as well. Each Pareto point represents a mapping of the four tasks to the system resources as well as their execution scheduling in these resources.

During the run-time scheduling phase the scheduler will select one of these Pareto points in order to meet the deadlines while reducing the energy consumption. For instance, let us assume that an application have to decode an image each 40 time units. If the Motion JPEG decoder is the only application that must be executed the scheduler will choose the point at the Pareto curve that consumes less energy, since all the points of the Pareto curve meet the deadline. Now let's assume that the

system must execute another task, for instance an OS routine, with an execution time deadline of 20 time units every 400 time units. Hence, one every ten executions of the JPEG decoder will overlap with the execution of this other task. When this happens the scheduler will select another Pareto point to adapt itself to the current situation. In this case, it will select a Pareto point with an execution time of 20 time units, and will schedule first the execution of the OS routine and, afterwards, the execution of the JPEG encoder. Hence, with the same 40 time unit slot, both tasks will meet their respective time deadlines although in the latter case the JPEG decoder will consume more energy.

In order to evaluate the benefits of this scheduling approach, we have compared the result obtained using the TCM scheduling approach with those obtained by a scheduler that tackles the same situation only at design time. Since in this case the system could not change the scheduling when the operative system routine overlaps with the JPEG encoder execution, the scheduler would have to choose between two options: either always selects the faster Pareto point to guarantee that the time constraints are met, or selecting the point that consume less energy and accept that when a operative system routine is executed the time restriction will not be met. Therefore, the design-time scheduler must choose between increasing the energy consumption, selecting a solution that works for the worst-case, or accepting a degradation of the quality of service of the JPEG application that will miss 10% of the deadlines.

If the degradation of the quality of service level is not acceptable, the designer would have to select the faster Pareto point. Assuming that an operative system

routine is executed once every ten iterations the medium energy consumption made with TCM will be: 11*0.1+5*0.9= 5.6 energy units. Whereas selecting always the faster point, the system will consume 11 energy units. Therefore, using the TCM design-time/run-time approach can lead to important energy consumption reductions, because the scheduler does not have to assume the worst-possible case. In addition, in complex systems the number of tasks will be large, and probably a worst-case approach will not be feasible.

4.9. Reconfiguration Overhead in TCM

TCM schedulers do not take into account the time needed to load a task in a given resource since in multi-processor platforms typically the time necessary to load a task is smaller than its execution time.

However, this assumption is not valid when, instead of addressing only conventional processors, the platform also includes reconfigurable units implemented using a FPGA. In this case, the time needed to load a task is bigger. For instance, loading a task which size is 10% of a FPGA VIRTEX-2 XC2V6000 requires 4 ms assuming a reconfiguration speed of 50MHz. The impact at the whole application execution of these 4 ms depends on with which frequency the reconfigurations are demanded. If the reconfiguration is required only from time to time, a latency of 4 ms is affordable. Nevertheless, as we have mentioned before the time execution of current multimedia application tasks is in the order of milliseconds. Hence, the reconfiguration latency can significantly degrade the whole system performance.

Turning again to the Motion JPEG decoder example, a typical deadline for this application is 33 milliseconds. If we want to load the four tasks into the four reconfigurable units, this process will consume 16 milliseconds. Therefore, it is clear that in order to meet the deadlines, the reconfiguration latency cannot be neglected. Besides, each reconfiguration also involves an import energy cost, since it typically involves moving a large amount of data from an external memory to the FPGA.

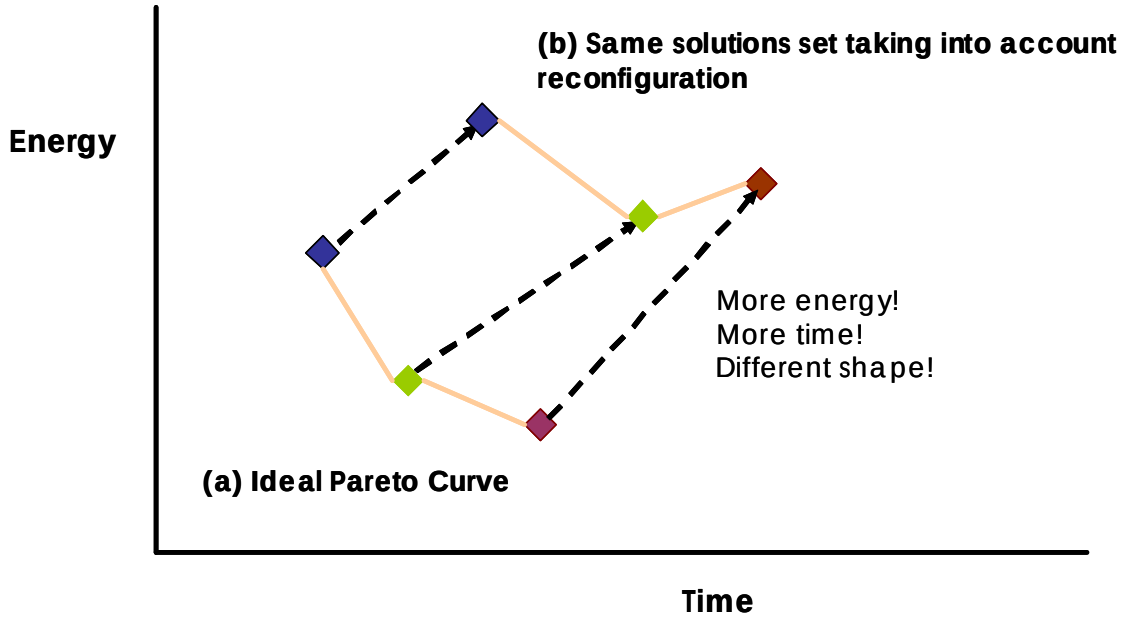


Figure 29. (a) Pareto curve builds without taking into account the reconfigurations. (b) Curve composed of the same solution that (a) curve but taking into account the penalization due to the reconfigurations.

Finally, as can be seen in Figure 29, the reconfigurations not only move the Pareto curve to a region with more energy consumption and larger execution time, but also some times they can even change the shape of the original curve. In this way, it is possible that the scheduler decide to change the Pareto point for a given task to reduce the energy consumption or the execution time, and the actual effect will be just the opposite due to the reconfiguration that this change entails. Therefore,

if the penalization due to the reconfiguration is not taken into account, the decisions made at execution time cannot be optimal. Moreover, it is not enough to take into account the reconfiguration latency when the scheduler is making its decision, but if the scheduler wants to use the reconfigurable HW efficiently, it must carry out an active policy that optimise the reconfiguration process reducing both the delays due to the reconfiguration latency and the energy cost. Otherwise, the reconfiguration only could be used for applications with very limited dynamic behaviour.

As we have explained in previous chapters, when executing recurrent task-graph those tasks that are already loaded previous executions can be executed again without carrying out the costly reconfiguration process. Hence, if the reuse percentage is high, the penalization due to the reconfigurations will be small.

Nevertheless, the greatest advantage of the TCM scheduling approach is that the schedules change in order to adapt to the run-time situation. To do this the scheduler chooses each time the most convenient Pareto point. As each point represents a different task mapping, it will not be always possible to reuse the configurations previously loaded. Besides, in order to take full advantage of the reconfigurable HW, the number of tasks executed in the reconfigurable units will be probably greater than the number of units. Hence, some tasks will overwrite to others, and therefore it will not always be possible to reuse a task even if it is assigned to the same resource in two consecutive iterations.

In order to support the task reuse and minimize the penalization due to the reconfigurations of those tasks that can not be reused, we have developed a set of modules that, collaborating with the TCM task scheduler, can drastically reduce the

overheads due to the frequent reconfigurations. These are discussed in the subsequent chapters.

Chapter 5:

Reducing the Reconfiguration Overhead in Systems with Conventional Configuration Memory Hierarchies

The task schedulers developed for multiprocessor platforms, as the TCM schedulers discussed in chapter 4, do not normally include in their estimations the time needed to load a task into a processing element. Indeed, when the task code is already loaded in a local memory, this time can be neglected. However, as it has been shown in the previous chapter, if some of the processing elements are RUs, the time needed to load a task, is not only significant, but in some cases it can be even higher than the task execution time for one task instantiation. Hence, estimations that

do not take into account the overhead due to the reconfiguration on such platforms, will be very inaccurate and will lead to clearly suboptimal decisions.

As we have mentioned in chapter 1, to reduce this lack of precision we have developed different modules that analyse the initial sequence of scheduled task-graphs selected by the scheduler and that include the overhead due to the tasks loaded in the RUs. Moreover, these modules also take decisions applying different optimisation techniques in order to minimize the impact of these overheads, in the given schedule. The run-time scheduling flow proposes in this work is presented in Figure 30.

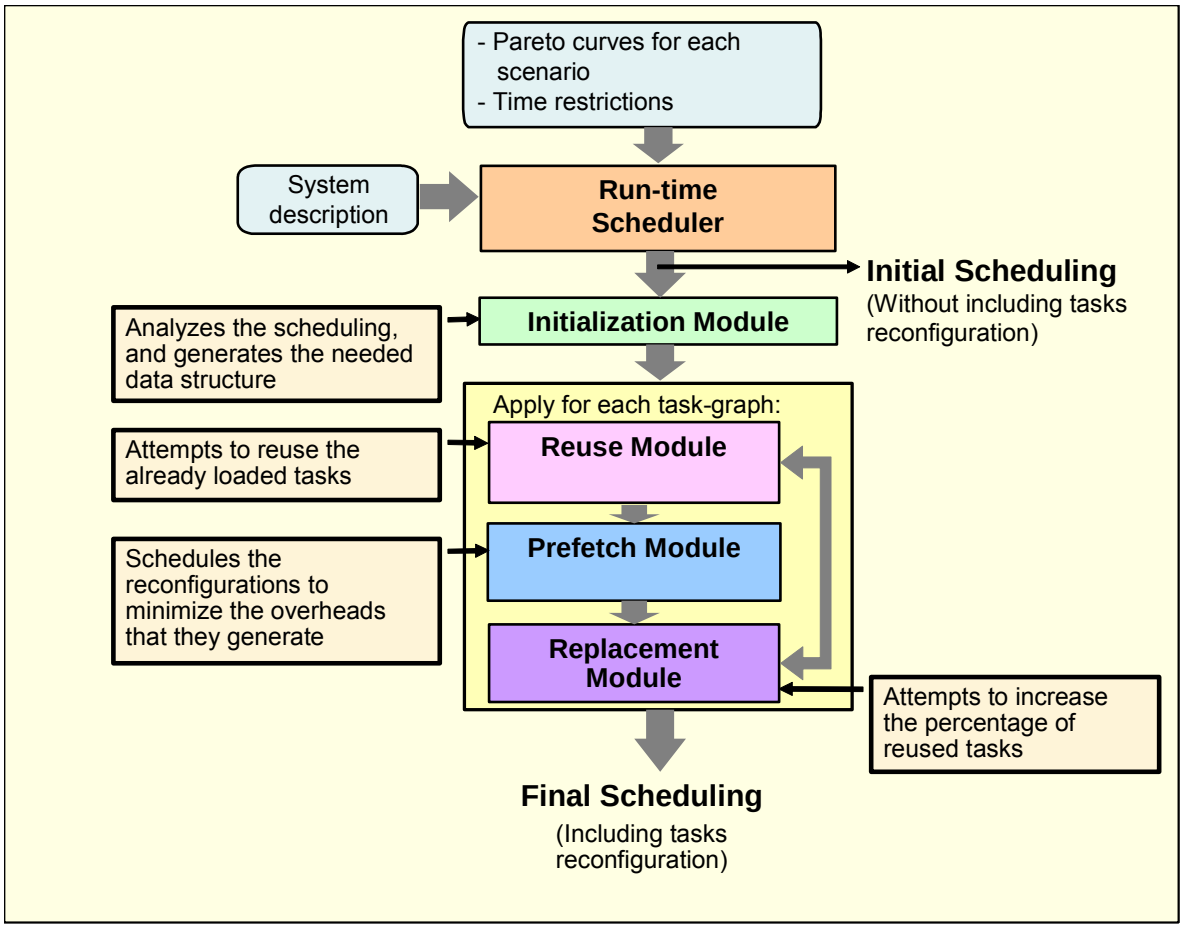


Figure 30. Proposed run-time scheduling flow.

Starting with the selected sequence of scheduled task-graph, the initialization module carries out an analysis and generates the data structures needed by the other modules.

Afterwards, the remaining modules apply several optimization techniques sequentially for each task-graph, following the execution sequence order established in the initial schedule. Three modules are applied to each task-graph, namely, the reuse module, the prefetch module and the replacement module.

The reuse module takes advantage of the possibility of reusing tasks that are executed periodically (this is very frequent, for instance, in multimedia applications). Hence, some tasks are loaded once and executed many times. This module checks if some of the tasks assigned to the HW RUs are already loaded. If that is the case, the task can be executed directly, without being loaded again into the RU.

The second module schedules the reconfigurations of those tasks that cannot (yet) be reused. This includes the first instantiation of a repetitive task. This schedule attempts to hide the loading latency by applying a prefetch technique that schedules, if it is possible, all the reconfigurations in advance. Therefore those configurations that can be prefetched do not introduce any execution-time overhead in the critical path.

Finally, the replacement module decides in which RU each task is going to be loaded. The aim of applying a replacement policy for the loaded configurations, is attempting to maximise the percentage of reused configurations. Hence, each time that a new task must be loaded it attempts to predict which of the loaded task has

less possibilities to be executed in the near future, and that task is the one replaced by the new one. This module takes into account the initial schedule in order to optimise its decisions.

These three modules are applied at run-time, since if the applications have a high dynamic behaviour, it is impossible to predict which tasks are going to be reused at design-time, and without this information it is not possible to schedule the load of the remaining tasks efficiently. However, these techniques also analyse the task-graphs at design-time in order to extract some useful information and to reduce the complexity of the run-time optimization process. Hence, they are based on a hybrid design-time/run-time approach, similar to the TCM task scheduler itself [MMSY07].

The next sections describe in detail the interaction of each one of these modules with the overall task manager and the optimization techniques that they apply.

5.1. Interaction of the Optimization Modules with the Task Manager

These modules are developed to reduce the reconfiguration overheads and they are not replacing the task scheduler. But they have been designed to collaborate with any existing task scheduler for heterogeneous multiprocessor systems in order to provide support to target reconfigurable units as a regular processing element in the system.

To evaluate these modules we have integrated them in the TCM scheduling environment described in the previous chapter. Nevertheless, it is possible to use other scheduler approaches as long as they are first adapted to generate the information that our modules need. Figure 31 depicts the functionality that a task scheduler has to include to be able to work with these modules. Basically, the scheduler has to interact with the application to know the task-graphs that have to be executed, the dependences between them, and the temporal restrictions. Taking into account this information, the scheduler has to decide in which processing element each task is going to be executed, and also their execution order.

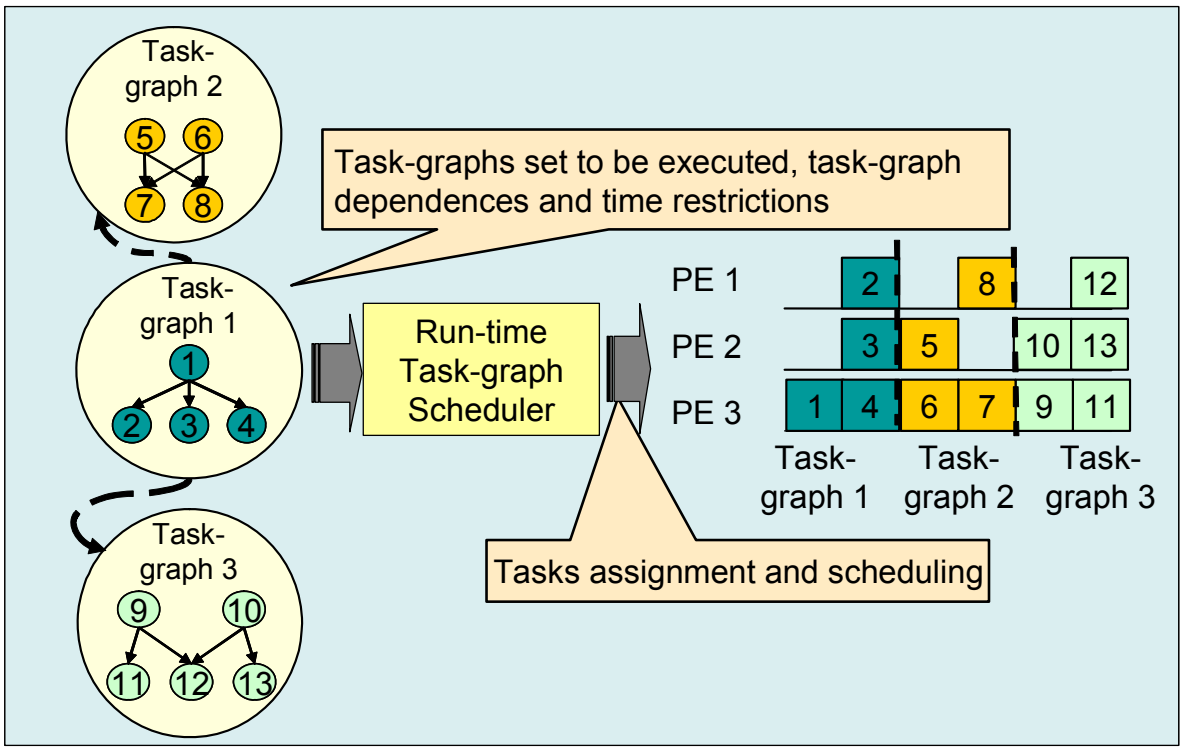


Figure 31. Task-graph scheduler execution example.

Besides, to be compatible with these modules the task scheduler has to use the same format that TCM uses to represent each task-graph, that is, a graph of

tasks where the dynamic behaviour is quite limited. All the strongly dynamic (non-deterministic or event-driven) behaviour is located in the constructs which are gluing together the separate task-graphs [HCDV03]. The information of the schedule and the task assignment selected by the scheduler is annotated in this task graph.

As we have mentioned before several graphs can be generated for the same task-graph, in the way that each one of them represents the task-graph execution under certain conditions. Each one of these graphs is called a system scenario [GPHV09], or scenario for short. The use of different scenarios allows including a certain level of dynamism (related to data-dependent conditional and loop behaviour) inside each task-graph. Using these scenarios is acceptable to define task-graphs with conditional structures or while loops depending on external data. At run-time the scheduler must take into account these external data and select the proper scenario. Nevertheless, the main part of the dynamic behaviour of the application must remain outside of the task-graphs. A dynamic application has to be represented as a set of graphs that interact among them, responding to a set of external events. To model the application behaviour at this level TCM uses a specific format [ThCa00] that is an extension of Petri nets [Mura89]. The usage of this format is not necessary in our modules because the task scheduler is the one that has to interpret this format, and once the task scheduler has taken his decisions, no other module uses the original representation, but only the separate annotated task-graphs. Another task scheduler different than TCM that use another model to represent the overall applications will be also valid as long as it uses a similar internal task-graph representation, i.e. direct acyclic graphs.

It is also not strictly necessary that the scheduler works in two phases, one at design-time and another at run-time. Nevertheless, this a highly recommendable option when it works with dynamic applications. Since, in this case, on one hand, due to the dynamism at least part of the decisions have to be taken at run-time, and on the other hand, if all the decisions are taken at run-time either the scheduler would generate an excessive overhead, or it would have to apply extremely simple heuristics that will not be able to take sufficient advantage of all the scheduler freedom and optimisation opportunities.

In systems with highly deterministic applications and no dynamism, it would be possible to use the scheduler at design-time. The modules presented in these PhD can then be applied without problem only at design-time and interact with this scheduler. Nevertheless, to work only at design-time modules with higher complexity could be used, since the time restrictions would be not too demanding.

However the main target of this thesis is not to develop techniques that ensure optimal solutions at design time, but heuristics that come close to the optimum in a short time, and that hence can be used at run-time providing good solutions while generating affordable cycle penalties.

5.2. Reuse module

This module receives as input data the sequence of scheduled task-graphs selected by the scheduler, that is, a graph where each task has been assigned to one of the processing units of the system in a specific order. Its aim is to identify if the

tasks assigned to the RUs must be loaded in order to execute them, with their respective penalty, or if it is possible to execute them directly because they were already loaded in a previous iteration and since then they have not been replaced.

If several tasks of the same task-graph are assigned in a certain order to the same RU, the first one it is called the initial task. The reuse module will never change the execution order of the tasks, thus if several different tasks of the same task-graph have been assigned to the same RU, the reuse module only can attempt to reuse the first one (the initial task). Since for example, the second task is presented in the system, it would have been overwritten to execute the initial task. Hence when the reuse module checks which tasks can be reused, it only needs to take into account the initial tasks.

Despite of this module not changing the tasks execution order, it will be useful if it can take advantage of a small level of flexibility when tasks are assigned to different RUs. For example, a task that remains loaded at the RU 1 could be assigned in the next iteration to the RU 2 and it would be impossible to reuse it. To solve this, the task scheduler works with virtual addresses regarding to the physical RU. Later the reuse module partly carries out the identification of these virtual units with the physical elements. This process will be finalized by the replacement module. Following with the example, if it is detected that the task assigned at virtual unit 2 is already available in the physical unit 1, the reuse module will identify the virtual unit 2 with the physical unit 1, and the task will be reused. Each physical unit is identified with a virtual unit and vice versa. This identification is done once for each task-graph

execution. Obviously, the number of virtual units and the number of the physical units has to be the same.

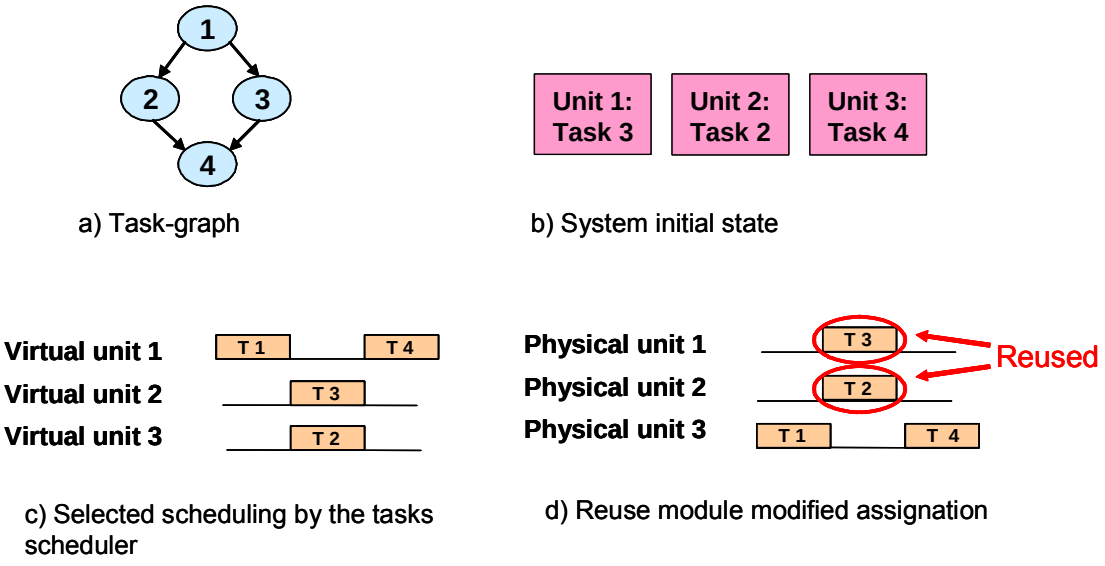


Figure 32. Reuse module execution example. T i: Task i execution.

Figure 32 depicts an example of how the reuse module works. Firstly, the task scheduler selects a scheduling where tasks are assigned to the virtual HW units. Afterwards, the reuse module analyzes this scheduling to identify if some of the initial tasks (tasks 1, 2 and 3 for this example) can be reused. In order to do this, it checks if some of them are already loaded in the RUs. In this example tasks 2 and 3 are loaded at unit 2 and 1 respectively, hence, the reuse module identifies the virtual unit 2 with the physical unit 1, and the virtual unit 3 with the physical unit 2. Those tasks that cannot be reused have to be loaded at the reconfigurable device. The prefetch module will decide when each task has to be loaded in order to hide as much as possible its reconfiguration latency. The replacement module will decide in which RUs those tasks are loaded that cannot be reused. In this example this is a trivial decision

since only one virtual unit and one physical unit to be assigned are present; hence the only option is loading tasks 1 and 4 into unit 3.

The aim of this reuse module, which is shared for the prefetch and the replacement modules, is not to generate a new scheduling at run-time, but attempting to minimize the reconfigurations impact while retaining the scheduling selected by the task scheduler. In our approach most of the computations related with the task schedule can be carried out at design-time using virtual units. When all the RUs are in the same reconfigurable device that works with a similar approach to the ICN model, assigning a task to Unit *n* instead of the Unit *m* has a minimum impact in the system. Firstly, because in ICN all units are identical. Secondly, because communications are carried out correctly, regardless where the tasks are finally loaded, as long as the address tables are properly updated. Finally, because in ICN communications are carried out using a segmented routing technique, call wormhole routing [DuYN97], and with this approach the latency of a communication almost does not depend on the distance. Hence, it is possible to estimate the communication time even though the physical units where tasks are going to be executed are not known yet. A drawback of this approach is that it is not possible to estimate precisely the energy consumption due to the communication between virtual RUs in HW. Indeed, for energy consumption the distance between the communicated units is important. More distance entails that more units are used to carry out the routing, and this implies an increment on the energy consumption. Hence, to estimate the energy consumption it is necessary to use a dedicated estimation heuristic, such as using the average distance for all the estimations.

The reuse module pseudo-code is depicted in Figure 33. It has a complexity of O(*NU***I*) where *NU* is the number of RU and *I* is the number of initial tasks of the task-graph. Generally *NU* and *I* are small and hence the execution time overhead of the reuse module is quite small in practice.

```
for (i=0; i< I; i++)
{
        found := 0;
        j := 0;
        while (not found) AND (j<NU)
        {
                found := try_to_reuse(i, j);
                if (found) { assign(taski, unitj) }
                j++;
        }
}
```

Figure 33. Reuse module pseudo-code.

5.3. Prefetch module

Current FPGAs do not support simultaneous reconfigurations. Thus, only one reconfiguration can be carried out at a time. Hence, if after applying the reuse module several tasks cannot be reused, they must be loaded sequentially. Since in the schedule selected these loads are not included, their execution order will have to be decided at run-time. The target of the prefetch module is to select the optimal sequence of reconfigurations.

The input of the prefetch module is a task-graph schedule where the reusable tasks have been already identified. This module analyses the schedule and decides

the reconfiguration sequence attempting to reduce as much as possible the execution-time penalty due to these reconfigurations.

A typical conventional load module would attempt to start loading a task when this task must begin its execution. Therefore, the execution of these tasks will be delayed due to the reconfiguration latency. This delay will be at least the time needed to load a task (we have been working in our experiments with 4 ms that is the time needed to reconfigure 10% of a Virtex2 v6000 with a clock frequency of 50 MHz), but it could be even higher if another task is being loaded onto the FPGA at the same time. For example, if four tasks are scheduled to start their execution at the same moment, the first task loaded will have a delay of 4ms, but the forth task will have a delay of 16ms, since its load could not start until the load of the previous three tasks has finished. These delays may not be acceptable for a system targeting current multimedia applications.

In order to reduce this overhead, this module applies a prefetch polity that, instead of waiting until the task has to start its execution, attempts to load the task as soon as possible, in order to hide the reconfiguration latency. Prefetch techniques have already been used to load data and instructions in advance in conventional processors [CaKP91].

This prefetch technique attempts to overlap the loading of the configuration of the tasks with the computation of other tasks in order to hide its latency. A very simple example is depicted in figure 34, where 4 tasks must be loaded and executed on a device with 3 RUs. 34(a) depicts the ideal case when no overhead exists. 34(b) presents the best on-demand schedule if the configurations are not prefetched. Thus,

the tasks are loaded when they are ready for being executed. As it can be seen in this example, since current FPGAs do not support multiple simultaneous reconfigurations, task 3 cannot be loaded until the loading of task 2 had finished. In 34(c), a schedule that applies prefetch is depicted. This time tasks 2, 3 and 4 are loaded as soon as possible, even when they are not ready yet for starting their execution. Thus, they introduce almost no time-penalty. Only the load of the task 1 penalizes the system execution time. In this example it can be seen how the prefetch module attempts to schedule the task loads in parallel with the execution of the previous tasks. Only for the first task it has not been possible, since the module did not have the needed flexibility to prefetch this task.

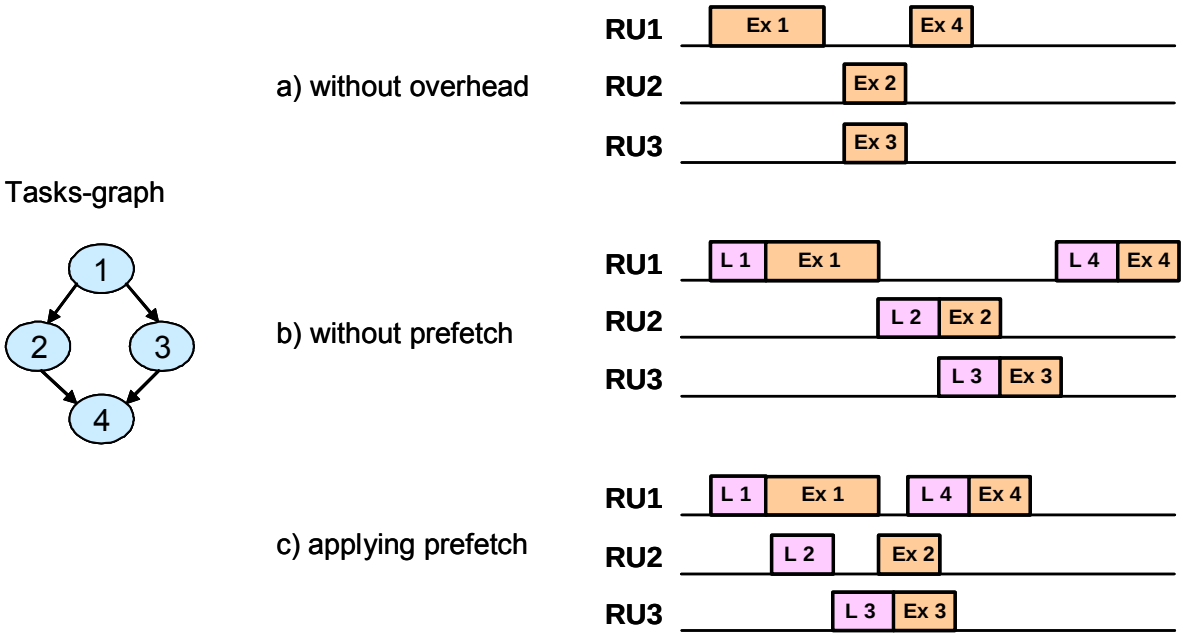


Figure 34. Example of configurations prefetch, on a platform with 3 reconfigurable units. L i: Task i load, Ex i: Task i execution.

Clearly, this powerful technique leads to significant execution time savings, widely minimizing the overhead due to the task loads. Unfortunately, deciding the

best order to load the configurations is not a trivial problem. Indeed its complexity grows exponentially with the number of tasks. In addition, the reconfiguration schedule must take into account that a task cannot be loaded if the previous tasks assigned to the same RU have not yet finished their execution.

Moreover, to complicate the problem even more, in order to apply this technique in conjunction with the configuration reuse technique, the schedule of the reconfiguration must be established at run-time. The reason is that the number of configurations that must be loaded depends on the number of configurations that can be reused and typically this number will differ from one execution to another if the system behaviour is non-deterministic. Thus, the set of tasks that are going to be reused is not known at design-time. Hence, the prefetch module has two objectives. On one hand, it has to schedule the load of a set of configurations minimising the run-time overhead due to these loadings. And on the other hand, it has to find a near optimal schedule while generating a very small run-time penalty into the system. To achieve these two goals, we have split the scheduling process in two phases, one applied at design-time and the other at run-time.

5.3.1. Design time computations

To carry out a wide number of the calculations at design-time allows reducing significantly the penalty of the prefetch module at run time.

Nevertheless, it is not possible to carry out all these calculations at design-time, since the optimal schedule depends on the number of tasks reused, that at the same

time depends on the FPGA state. This state varies during the execution, and as we want to work with dynamic applications, it is not possible to foresee these changes at design time.

The data available at design-time are the task-graphs and the features of the platform where the application is going to be executed. With this information the task scheduler generates at design-time a set of schedules (in TCM it means a Pareto curve for each task-graph scenario) where the tasks are assigned to the different resources of the platform. Analyzing each one of these schedules it is possible to determine how critical each task is in the execution of the task-graph. Hence, the first task in the critical path is the most critical task of the task graph, since if it is delayed all the tasks in the critical path will be delayed. However, the execution of those tasks that are not in the critical path can be delayed up to a certain point without increasing the total execution time of the task-graph. Hence, we can say that certain tasks are less critical, or less important, than others for the execution time of the global task-graph point of view. These tasks have a relative range in which they can be moved without incrementing the execution time of the global task-graph.

So, at design time every task of each task-graph is tagged with a weight that represents how critical is its execution. These weights are computed performing an ALAP (as late as possible) scheduling. Thus, they represent the longest path (in terms of execution time) from the beginning of the execution of the task to the end of the execution of the whole task-graph. With this criterion the first task in the critical path has always more weight than the others. In figure 35 it can be seen how these weights are computed for a simple task-graph.

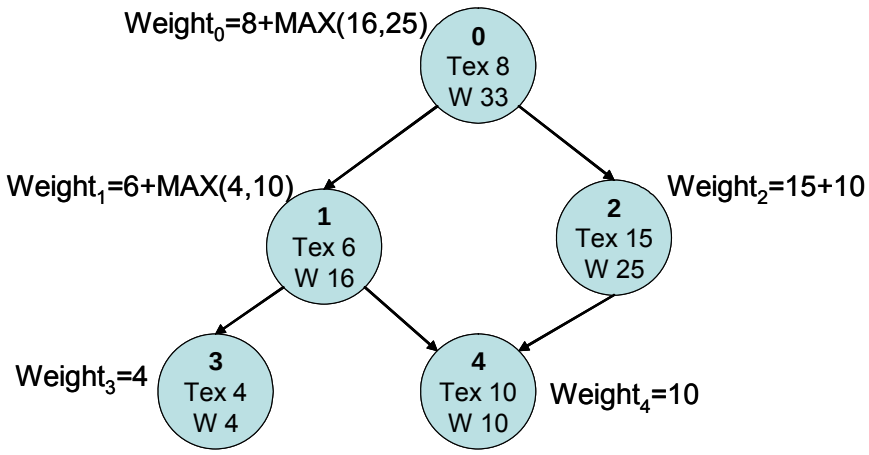


Figure 35. Weight calculation for a task-graph. Tex: task execution time. W: task weight.

If a node has no successors, its weight is its own execution time. Otherwise, its weight is its own execution time plus the weight of the successor with the greatest weight.

In order to compute the weight of each task, the algorithm only needs to know the task-graph structure and the execution time of each task. Hence, it can be carried out at design-time, and afterwards its results can be used during the run-time phase without generating any execution-time penalty.

However, it must be taken into account that the execution time does not only depend on the task-graph but also on the task assignment. Hence, the weighs must be computed for each different assignment selected by the design-time scheduler.

In our scheduling flow, we receive as input an already scheduled task-graph, where all the tasks have been assigned to a RU. Hence, we must take into account all the information of this initial schedule when computing the weights. Figure 36

depicts the same example as figure 37, but this time adding the information of a given schedule.

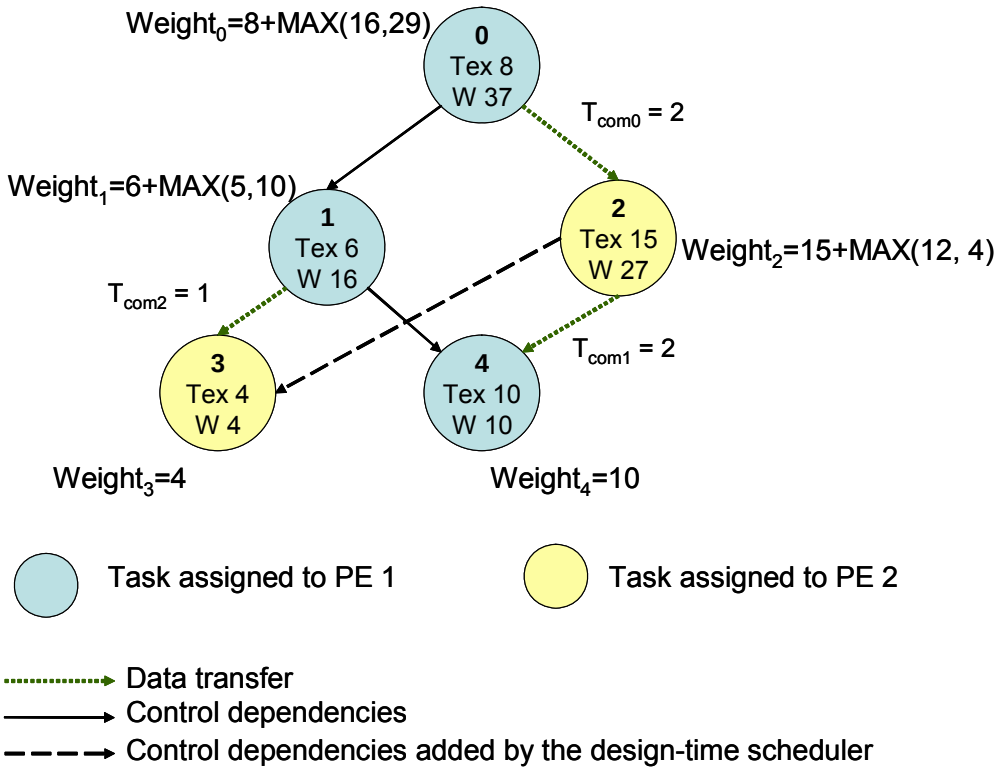


Figure 36. Weight calculation for a task-graph taking into account the information of the tasks scheduling obtained by the design time scheduler. Tex: task execution. W: weight of the task. Tcom: time needed to carry out a communication.

In this example the tasks have been partitioned between two different PEs (for instance two RUs). Moreover, each PE uses its own private memory during its execution. Hence, when two tasks assigned to different PEs need to share some data, this data must be sent from one PE to the other. Since the time needed for these communications is significant, it is included in the task-graph (in the edge corresponding to each communication). In this example, it is assumed that tasks assigned to the same PE can share the data in local memory without any cost.

The design time scheduler has also added a new edge in the task graph in order to impose the execution order in PE2. Due to the existing dependencies just one feasible execution order exists for PE1, hence no new edges have been added. If needed the design time scheduler can introduce synchronization dependencies, data-access optimisations or any other constraint needed. This information is always included when computing the weights. Since this phase is carried out at design-time, it does not generate any run-time penalty.

Once again, it must be remarked that a task-graph is used to represent mostly static behaviour. Hence, it is possible to compute the weight of each task based on accurate profiled information.

5.3.2. Run time computations

At run-time the reuse module has analysed which tasks can be reused and has annotated this information in the task-graph. In addition, this task-graph also contains the information of the schedule selected and the weights of each task. With this information, the prefetch module has to decide in which order tasks have to be loaded, in order to reduce as much as possible the impact of task load latency in the global execution time. To achieve this goal we have developed a simple heuristic based on list-scheduling [YaGe93]. Once the scheduling has been carried out, the prefetch module updates the initial ideal schedule selected by the task scheduler (which does not include the reconfiguration latency) with the information regarding the reconfigurations.

The heuristic starts from the root task of the task-graph, i.e. the one that has no predecessors. If several root tasks are present, a dummy root task must be added. Then it continues reading its successors as long as they do not demand any reconfiguration (either because they have been assigned to a processor or because their configurations are reused since they were loaded in a previous iteration). If a task needs to be loaded, a reconfiguration request is stored in a sorted list. The list is sorted using the point of time when the reconfigurations have been requested. When it is impossible to continue, one of the requests is selected according to the following criteria:

- If, at a given time *t*, when no reconfiguration is being carried out, just one configuration is ready to be loaded, this configuration is selected. A configuration is ready if the previous task assigned to the same reconfigurable unit has already finished its execution.
- Otherwise, when several configurations are ready, the configuration with the highest weight is selected.

Each time the scheduler selects one reconfiguration, following the previous criteria, it is scheduled as soon as possible. A task can be scheduled only if it meets two conditions. Firstly, the previous task executed in the target unit must have already finished its execution. Secondly, the previous reconfiguration has been already completed.

```
prefetch_heuristic ()
{
        schedule_update (root_node, &reconfiguration_list);
        for (i=0; i < Nl; i++)
        {
                node := load_select_&_schedule (&reconfiguration_list);
                if node.ready {
                        schedule_update (node, &reconfiguration_list);
                }
        }
}

schedule_update (node, &reconfiguration_list)
{
        schedule (node);
        for (i=0 ; i<number_of_successor (node); i++)
        {
                if (node.successor[i].loaded) AND (node.successor[i].ready)
                {
                        schedule_update (node.successor[i], &reconfiguration_list);
                }
                else if (node.successor[i].ready)
                {
                        ask_load (node.successor[i], &reconfiguration_list);
                }
        }
}
```

Figure 37. Task load scheduler pseudo-code. Nl: number of tasks to be loaded.

Figure 37 presents the pseudo-code of this heuristic that is basically a 'for' loop which is executed as many times as tasks have to be loaded. The process starts analyzing the root node (the one without predecessor). Then the recursive function *schedule_update* analyses the successors of the root node successors attempting to reach the leaf nodes. This function updates the schedule of those nodes whose predecessors have not been scheduled yet and that do not have to be loaded. Each time that a node has to be loaded before starting its execution, i.e., when it has been assigned to a RU and it can not be reused, *schedule_update* carries out a load request using the function *ask_load* that stores it in the *reconfiguration_list*, and it

does not continue analysing the successor yet until this load is scheduled. The *reconfiguration_list* is a sorted list where the sorting criterion is the point of time when the load of the node was requested. When the process cannot continue because the nodes waiting to be loaded block it, the *load_select_&_schedule* chooses one of the nodes in the *reconfiguration_list* and schedules its load. Once a task is selected to be loaded, the function *schedule_update* is requested again. It would attempt to schedule the recently loaded tasks and its successors, until it is blocked again by tasks that are waiting to be loaded. Then, a new task will be selected. This process continues until all the loads are scheduled.

When the load of a task is scheduled, its reconfiguration may introduce a delay into the initial schedule. The prefetch module includes these delays updating the initial schedule. Hence, the prefetch module generates a final schedule that includes the delays due to the reconfiguration latency.

The complexity of this heuristic is O(*Nl**log(*Nl*) + *Ne* + *Nt*), where *Nl* is the number of tasks that must be loaded, *Ne* is the number of edges, which represent dependencies in the task-graph, and *Nt* is the number of tasks in the task-graph.

Figure 38 presents a detailed example of the application of the prefetch heuristic for the task-graph presented in Figure 36. In this example the reconfiguration latency is 4 ms. The next two figures show, for the same task-graph, the ideal scheduling (assuming a load latency of 0 ms for illustration purposes) selected by the task scheduler and the final schedule after applying the prefetch heuristic. This schedule has been carried out assuming that no task can be reused, and that all tasks must be executed in two RUs.

The five tasks have to be loaded. At t=0 the load of the **task 0** is asked.

i = 0

Only one asked load exists: (0). The load of the **task 0** is scheduled from t=0 to t=4. The **task 0** is scheduled from t= 4 to t=12. **Com 0** is scheduled from t=12 to t=14. The loads of the **tasks 1 and 2** are asked in t=4.

i = 1

Two asked loads exist: (1, 2). The load of the **task 2** is scheduled from t=4 to t=8 since Weight2>Weight1. The **task 2** is scheduled from t=14 to t=29. **Com 1** is scheduled from t=29 to t=31. The loads of the **tasks 3 and 4** are asked in t=8.

i = 2

Three asked loads exist: (1, 3, 4). The load of the **task 1** is scheduled from t=12 to t=16 since in t=12 the tasks 3 and 4 are not ready (The previous tasks assigned to the same unit that tasks 3 and 4 have not finished their execution). The **task 1** is scheduled from t=16 to t=22. **Com 2** is scheduled from t=22 to t=23.

i = 3

Two asked loads exist: (3, 4). The load of the **task 4** is scheduled from t=22 to t=26 since in t=22 the task 3 is not ready (The previous tasks assigned to the same unit that task 3 have not finished their execution). The **task 4** is scheduled from t=31 to t=41.

i = 4

Only one asked load exists: (3). The load of the **task 3** is scheduled from t=29 to t=33 since in t=29 the execution of the previous tasks assigned to the same unit that task 3 finish. The **task 3** is scheduled from t=33 to t=37.

Figure 38. Step by step example of the prefetch heuristic assuming that no task is reused.

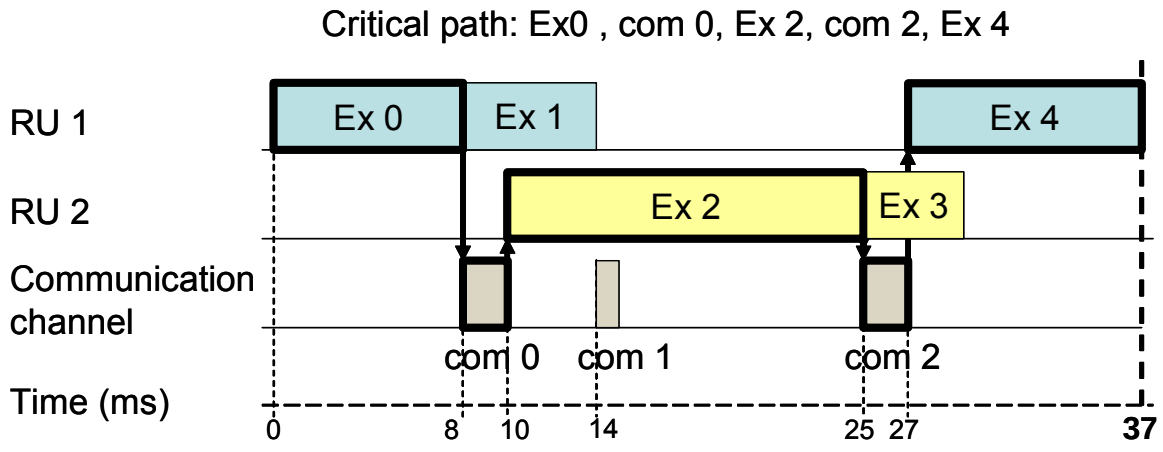


Figure 39. Ideal schedule selected by the design-time scheduler. The arrows represent the dependencies in the critical path.

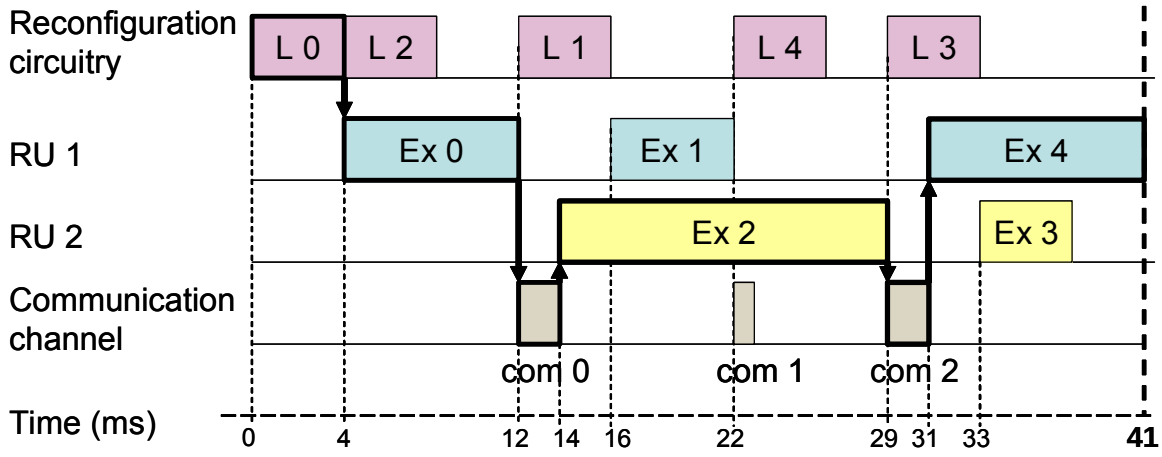


Figure 40. Schedule process presented in figure 38. The arrows represent the dependencies in the critical path.

In this example the schedule selected by our prefetch module is in fact the optimal schedule. If the final schedule is compared to the original one selected by the design-time scheduler, it can be observed that only the load of Task 0 generates a delay. This delay cannot be hidden because it is impossible to overlap it with the computation of previous tasks.

Although in many cases our heuristic finds the optimal schedule, this cannot be always guaranteed. In fact, this heuristic has been designed to look for near optimal schedules in an affordable time. To this end, the heuristic behaves greedily when just one task is ready for being loaded, and the reconfiguration circuitry is available. In this case, the heuristic always schedules the load of this task. This is a good decision most times, but it is not always optimal. In some cases, it is better not to schedule this task and wait until a more critical task is also ready for being loaded. One example of this case is depicted in figure 41. There, our heuristic decides to schedule the load of the task 1 at t=4 because at this moment it is the only task ready for being loaded. However, in t= 5 task 2 will be also ready. In this case it would be

better to wait for the task 2 and load it first, because it would reduce the total execution time in 2 ms.

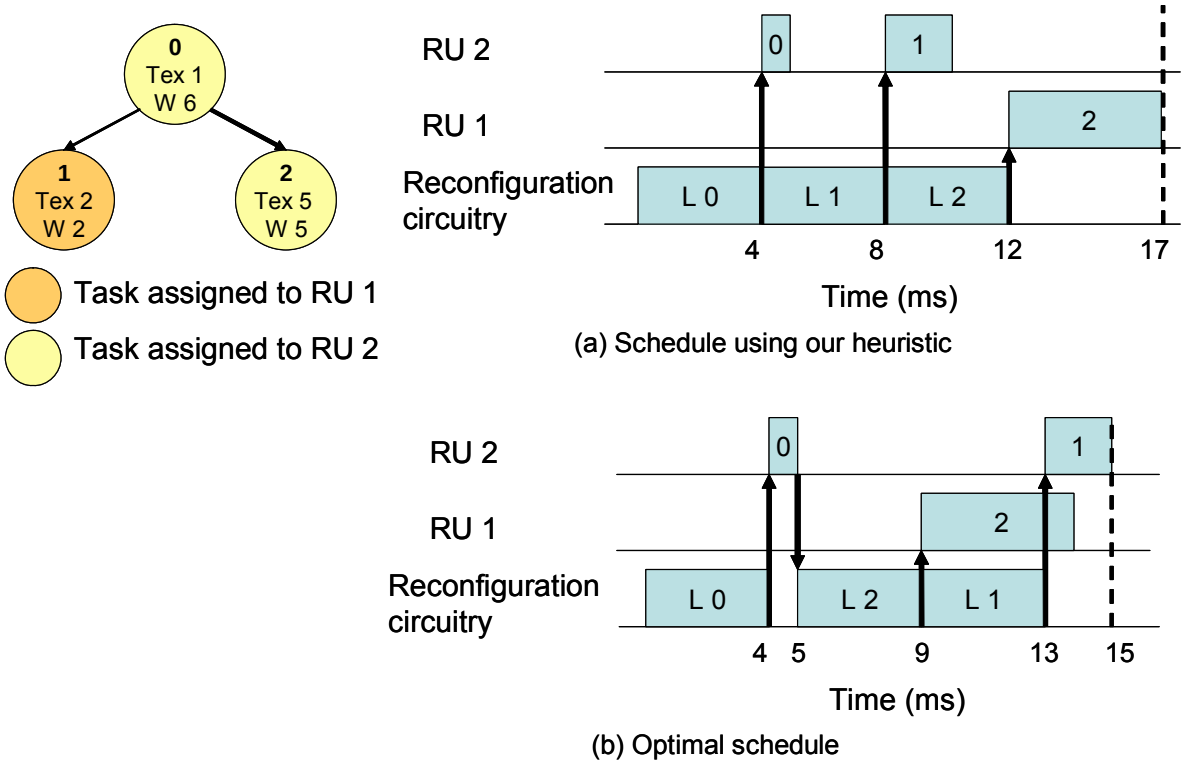


Figure 41. Schedule example where the scheduling heuristic does not found the optimal solution.

Since this heuristic is not optimal, we have implemented another prefetch scheduler that always guarantees that it finds the optimal scheduling. To accomplish this, it carries out an exploration of the solution space with a branch and bound algorithm (b&b).

To this end, the b&b algorithm starts evaluating one possible schedule of the reconfigurations, and from that schedule it explores all the possible changes. In order to reduce the execution time, during the search process the algorithm keeps a record of the best solution found, and when the execution time of a given schedule is worse than the best execution time found up to then, the b&b algorithm discards it in order

to save computational time. Thus, the algorithm guaranties that it will find the best possible schedule without carrying out an exhaustive design space search.

In order to compare these two options, we have generated a random set of graphs using the TGFF tool [DiRW]. It is a set of 100 tasks, with an average size of 20 tasks per task-graph. The b&b scheduler finds on average 10% better solutions than our heuristic. However, it needs 800 times more computational time to find these solutions. Moreover, while our heuristic has a O(*N**log(*N*)) complexity, the b&b algorithm has an exponential complexity. Hence, it will be not feasible to apply it to systems with a large number of nodes. This experiment confirms that our heuristic achieves almost optimal schedules while introducing a small run-time penalty.

5.3.3. Interaction of the prefetch module with the design and run-dime schedulers

Figure 42 presents how the scheduling decisions are split among the design-time scheduler, the run-time scheduler and the prefetch module. Thus, different decisions are taken at each level. These decisions are not orthogonal. In fact, each step imposes a set of constraints to the followings steps. Hence, the decisions taken by the design-time scheduler drastically influence the run-time scheduler and the prefetch module, and the decisions of the run-time scheduler reduce the scheduling possibilities available for the prefetch module.

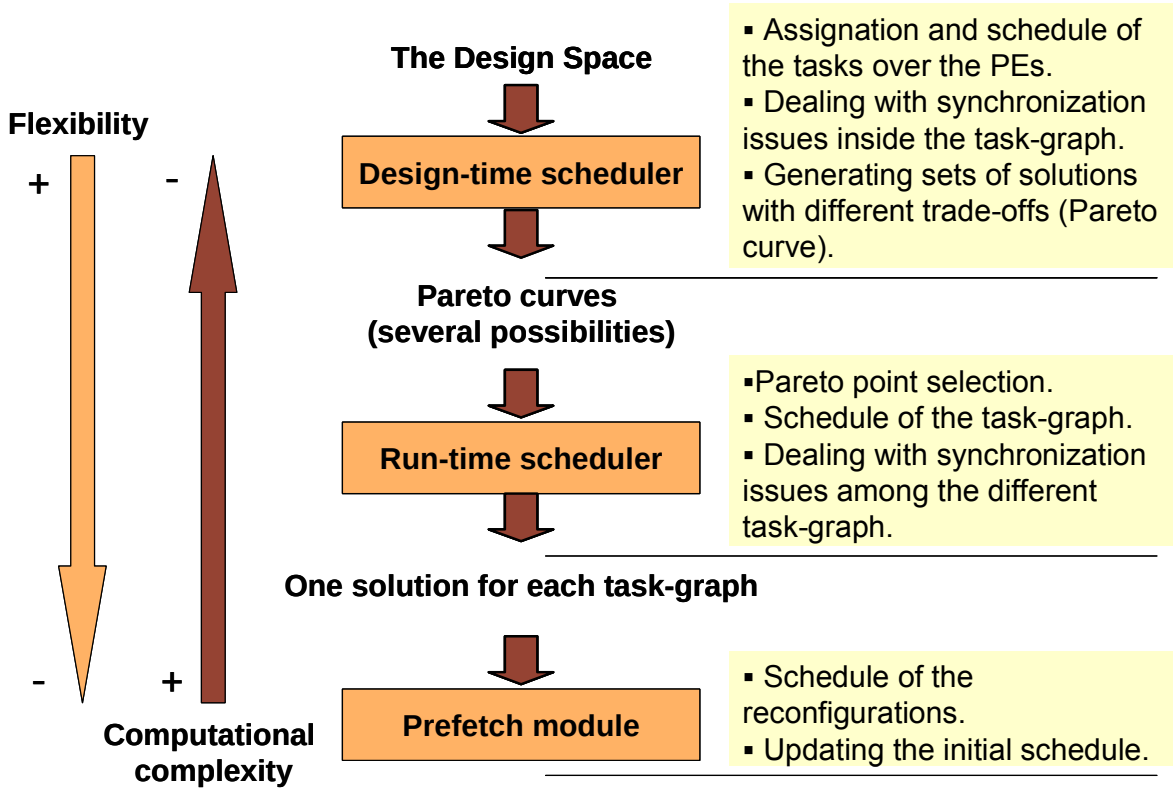


Figure 42. Scheduling flow.

The main reason for this loss of flexibility is that it is necessary to reduce as much as possible the complexity of the calculations at run-time. Thus, at design-time the scheduler has plenty flexibility, but this flexibility entails the exploration of a large design space. At run-time just a small set of the most promising potential schedules remains available to the run-time scheduler. Finally the prefetch module receives as input a single Pareto point, where the task assignment to the processing resources and the task execution sequence are already fixed. Hence, not too much flexibility is left for this module to carry out its optimisations.

Providing more flexibility to the prefetch module is a very simple task. It is just needed to move downward in the scheduling flow some of the decisions. For instance, the prefetch module could join the run-time scheduler, and compute for

each Pareto point the delay due to reconfigurations. Unfortunately, as the flexibility increases, so does the computational complexity. Since, instead of running the prefetch module once per task-graph, it would be run once for each Pareto point. As the prefetch module is executed at run-time, its complexity must remain always small in order to not introduce a great run-time penalty. Hence, another way of providing enough flexibility to the prefetch module is needed.

We have observed that if the design-time scheduler follows two simple rules, the degree of flexibility needed by the prefetch module to carry out properly its optimisations is maximised. These rules are:

- If the number of tasks assigned to dynamically reconfigurable HW is smaller than the number of RUs, assign each task to a different RU.
- Otherwise, if it is possible, do not assign two consecutive tasks of the critical path to the same RU.

The first rule maximises the possibilities of reusing the tasks of a task-graph when it is executed periodically, since at the end of their execution all of them remain loaded. In addition, it allows taking full advantage of the prefetch technique since all the tasks can be prefetched easily (all of them are initial in their respective RUs, hence they are always ready for being loaded). Figure 43 illustrates this idea. In this figure two different schedules for a given task-graph are proposed. Although four RUs exist, the first option assigns all the tasks to only two out of the four RUs. From the design scheduler point of view, this schedule is optimal in terms of execution time. However, when the prefetch module attempts to hide the latency of the

reconfigurations, only the one corresponding to task 3 can be hidden. The second option presents the same execution time from the design-time scheduler point of view. However, when the prefetch module attempts to hide the reconfiguration latency, much more flexibility remains, and it has found the way to hide 3 out of four reconfigurations. Moreover, if this task-graph is executed several times consecutively it will be possible to reuse all the reconfigurations, whereas in the schedule of option 1 it is just possible to reuse Task 3.

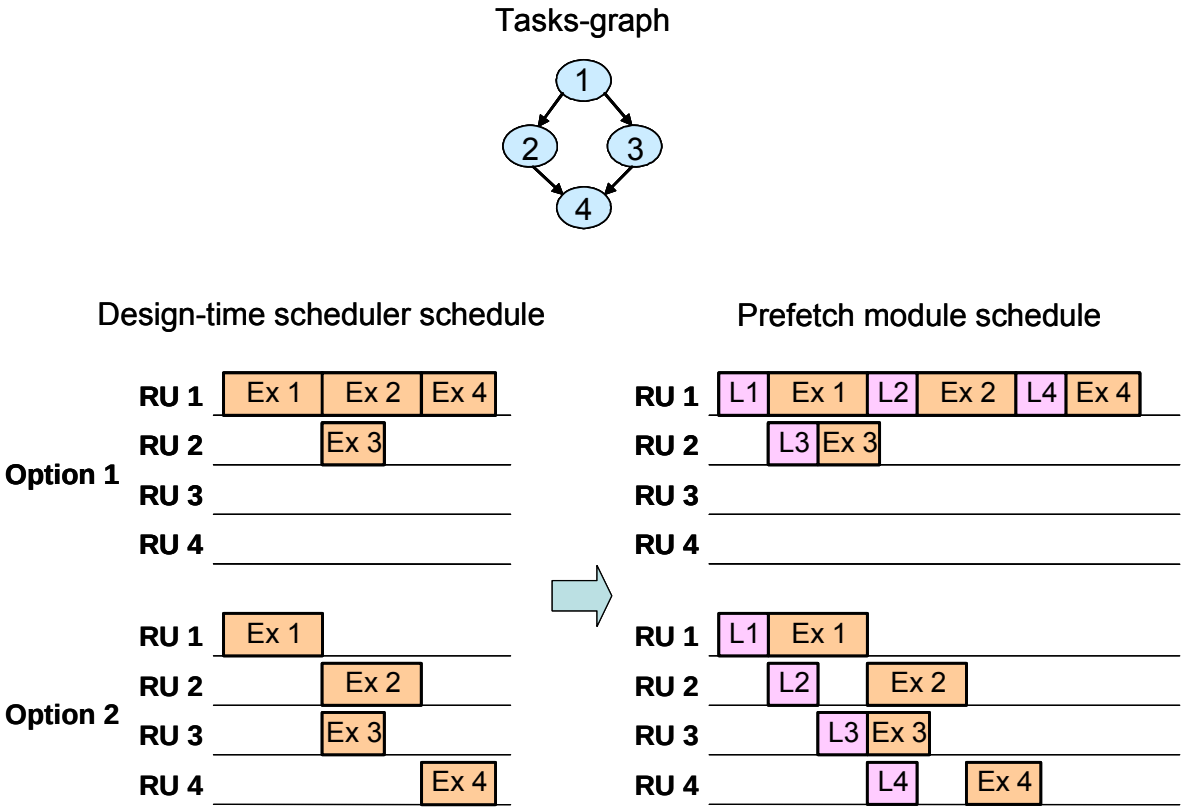


Figure 43. Comparison between two different schedules for the same task-graph.

Figure 44 presents, as in the previous example, two identical schedules, from the design-time scheduler. However, if the reconfiguration latencies are taken into account, the result obtained following the second rule is better. In the first schedule

the whole critical path has been assigned to the RU 1. This assignment drastically reduces the flexibility for the prefetch module. Thus, only the latency of one out of four reconfigurations can be hidden. On the contrary, in Option 2 the critical path has been divided between RU 1 and RU 2. Hence, it is feasible to load a task of the critical path while the previous one is being executed. As a consequence only the first reconfiguration penalises the overall execution time.

If the number of tasks inside a task-graph is bigger than the number of RUs, it is impossible to assign just one task to each RU. In this case rule two guarantees that at least some flexibility will remain when scheduling the tasks in the critical path. Since the global execution time is determined by the critical path, this is an important issue.

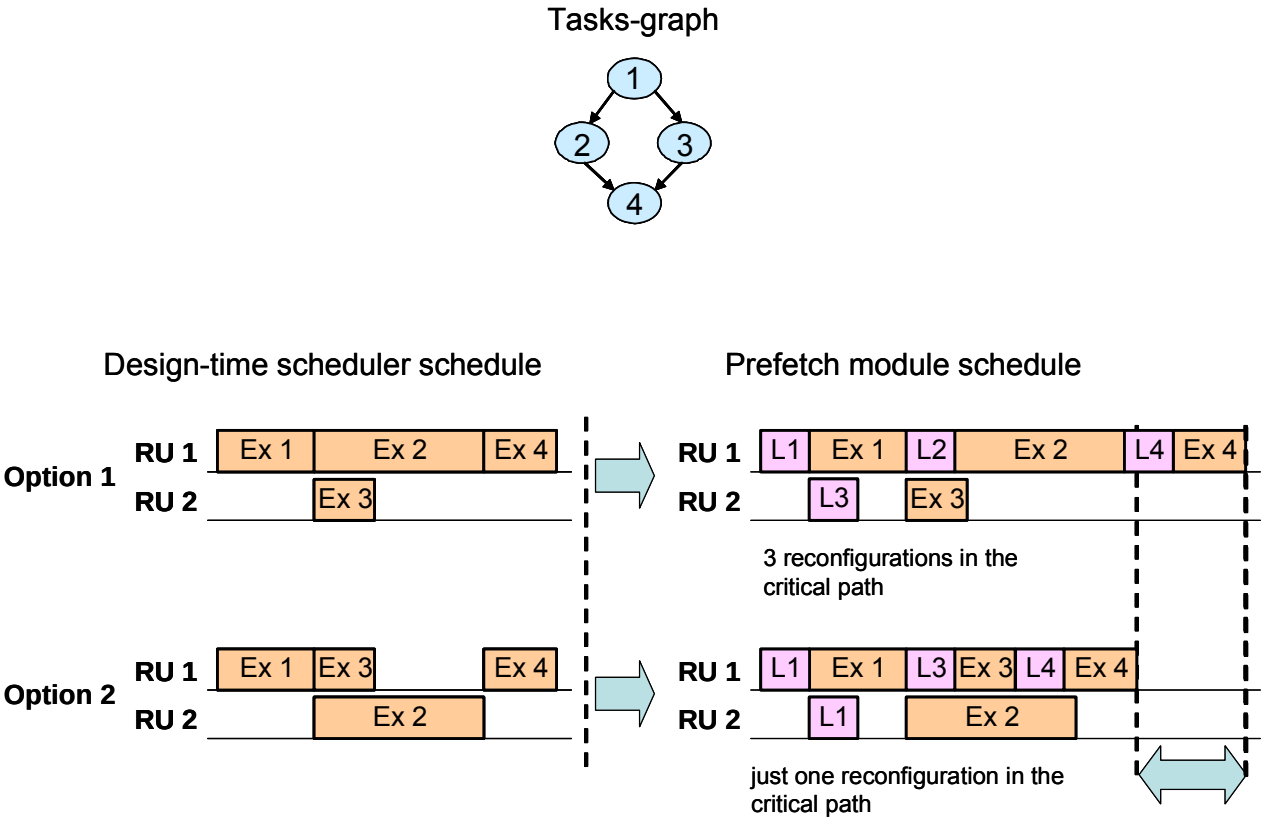


Figure 44. Comparison between two different schedules for the same task graph including the reconfiguration overhead.

5.4. Replacement module

The main goal of this technique is to reduce the number of reconfigurations. To accomplish it, this module attempts to maximize the percentage of reused tasks.

Multimedia applications are commonly composed of elements that are executed repeatedly, with different input information. Digital video applications and 3-D objects visualization applications are some common examples. Hence, it is likely that a task-graph executed at a specific moment would be executed again in the near future.

The reuse module identifies if a task can be reused, but it does not carry out any active policy to increase the percentage of reused tasks. Instead, the task replacement policy can positively affect this percentage. In order to minimize the number of reconfigurations, we have developed a replacement heuristic based on a well-known memory-page replacement strategy (LFD). This heuristic starts from a sequence of known events and from a system that can only load a finite number of elements. In the replacement module the events are the executions of certain tasks, and the elements are the tasks. When the system is full (tasks are loaded in all the RUs) and a task has to be loaded, one of the loaded tasks has to be replaced. The event sequence is analysed in order to select the task to replace. The element that is going to be used the latest in time is the one selected.

The LFD strategy is an optimal replacement technique, because it minimizes the number of loads [Bela66]. Unfortunately, in order to apply it properly we should be able to predict all the future events. Nevertheless, although we do not know the whole

future, once the run-time task scheduler selects the schedule of a task-graph we can use that information of the near future to simulate the LFD strategy. In our approach we will use the available schedules to extract information about the future and we will carry out predictions for those cases where not enough information is available.

5.4.1. Interaction of the replacement module with the design- and run-time schedulers

Figure 45 depicts the complete assignment flow of tasks to the RUs.

Initially the design-time scheduler searches the design space and selects a Pareto curve of solutions with different trade-offs for each task-graph. Each solution consists of a task assignment to the PEs and an execution schedule. Then, the run-time scheduler selects one of the Pareto points for each active task-graph. However, as it was explained in the reuse module section, in order to provide some flexibility for the reuse and replacement modules, both the design-time and run-time schedulers work with virtual addresses (called virtual RUs). Thus, a given virtual RU can be identified with any physical RU of the system.

The virtual RUs are identified with physical units in two steps. Firstly, the reuse module detects those tasks that can be reused and identifies the virtual units that contain them with the physical units where they were previously loaded. Afterwards, each time that a non-initial task is scheduled, the replacement module decides in which physical RU it is going to be loaded, applying its replacement policy.

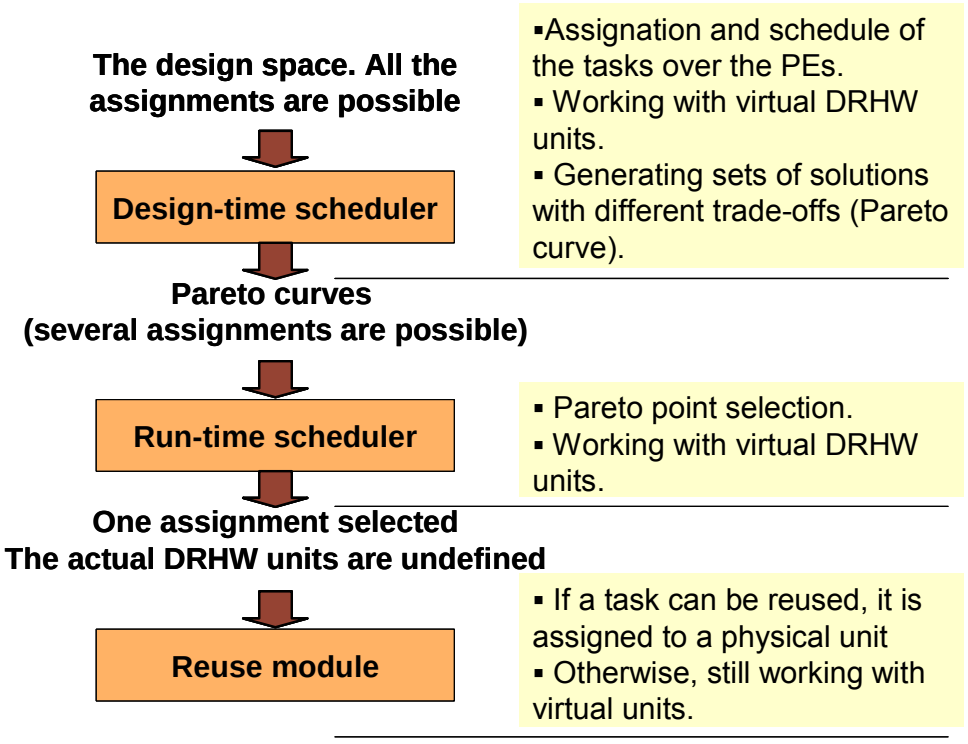


Figure 45. Tasks assignment process to PEs.

Once all the initial tasks have been assigned to physical RUs, it is not needed to call the replacement module for the remaining tasks, because these tasks are automatically assigned to the same RU as their corresponding initial tasks. Figure 45 illustrates this idea.

If the design–time scheduler assigns a set of tasks (in Figure 46 tasks 1 and 4) to the same virtual RU, the reuse and the replacement module must also assign these tasks to the same physical RU. Thus, if the initial task 1 is assigned to the RU 3, task 4 is assigned automatically to the same RU.

The replacement module does not change the initial assignment selected by the design-time scheduler. This reduces the flexibility of this module. But with this

approach the most computational intensive parts of the scheduling process can be carried out at design-time.

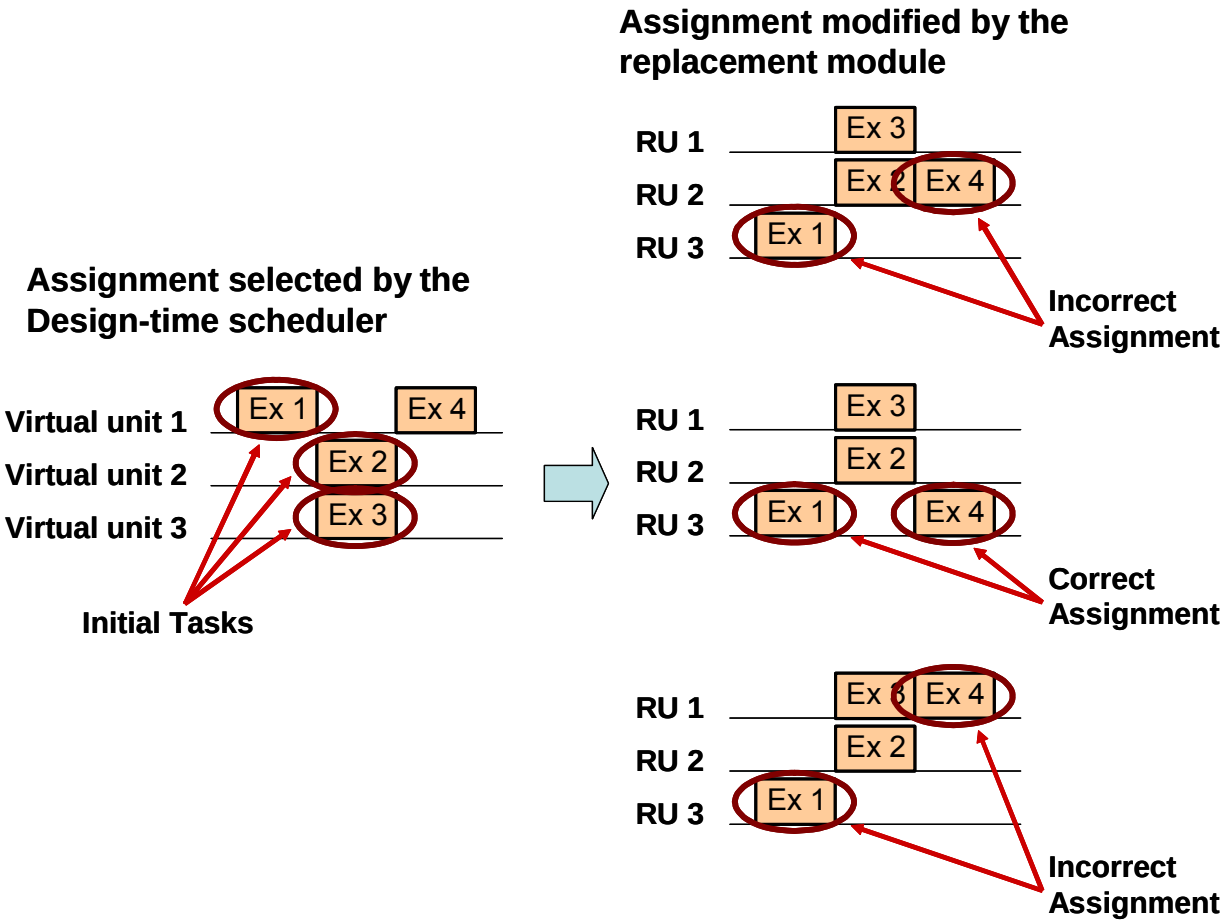


Figure 46. Examples of assigning the non-initial tasks to physical units.

5.4.2. Replacement heuristic

After the run-time scheduler selects its schedule, an initialisation module parses it. The goal is to identify which of the tasks that are currently loaded in some of the RUs are going to be executed again during scheduled time. This analysis is carried out for all the scheduled task-graphs at the same time in order to use all the information available.

After this analysis the RUs are divided in two categories, namely, reusable RUs and non-reusable RUs. The first RUs are those that hold a task included in the current scheduled sequence of task-graphs. The second category includes those tasks that are not currently scheduled, or are going to be executed in a non reconfigurable PE.

Each time that the load of an initial task is scheduled, the prefetch module invokes the replacement module to decide where to load the task. The replacement module takes this decision according to the priorities of Figure 47.

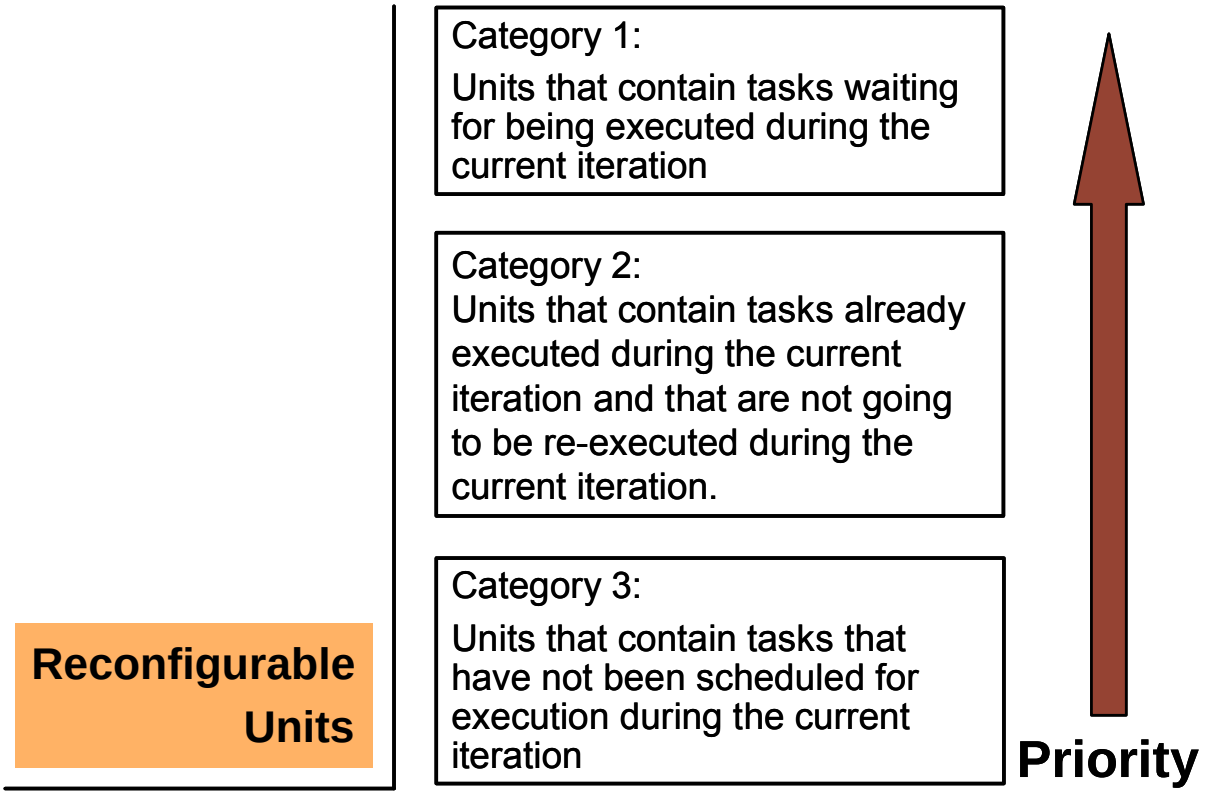


Figure 47. Priorities of the RUs.

As it can be seen, non-reusable tasks are those with less priority (category 3 in figure 47). Then, reusable tasks are divided in two categories. First, those tasks that

have not been executed yet are the ones with the maximum priority (category 1 in figure 47). And second, with less priority, those tasks that have been already executed, otherwise they still belong to the first category (category 2 in figure 47).

When a new task must be loaded, the task to replace is selected from the category with less priority that is not empty. It gives more priority to those tasks that belong to the first category. It agrees with LFD strategy, and therefore is an optimal decision if the goal is to maximise the reuse of configurations because if these tasks are not removed, they are going to be reused at least this period.

On the contrary, giving more priority to the second category than to the third category is not always optimal. Since, we do not really know if any of those tasks is going to be executed in the near future. Hence, it is possible that a task of the third category would be executed in the future before some of the tasks in the category two. However, we give more priority to the second category in order to exploit the temporal locality since those tasks executed recently have more chances of being executed in the near future. For instance, when a new object or image appears in a 3D game or in a video-decoding application it usually appears in more scenes, otherwise the user will not even notice it, hence all the tasks related with this new object will be executed again.

After selecting the category, it is necessary to choose the specific unit to load each task. Inside each category this selection is done in different ways:

1. To select a node of the third category, a simple random policy is applied to select which task must be replaced. Of course this is not an optimal policy, but

since most of the times, all the tasks assigned to this category are going to be replaced, it is not worthy to waste computational time applying a more complex policy.

2. If the third category is empty, the replacement heuristic selects an element from the second one. In this case, it is not possible to know which task is going to be executed further. Hence, to apply the LFD policy the replacement heuristic assumes that the next iteration is going to be similar to the current one. In this case the decision may not be optimal if the assumption is not correct, and a random heuristic could be used. However, we have observed that in practice this assumption is again taking advantage of the temporal locality that is normally available in our target domain, and it will provide results close to the optimal replacement heuristic.

3. If the third and second categories are empty, a task from the first category is selected following the LFD replacement policy. Therefore, the RU is selected with the task that is going to be executed further from the current point of time.

5.4.3. Implementation details

The replacement heuristic is implemented using three lists: Non-reusable units list, Replacement list and Temp list. These lists are generated by an initialisation module that parses the output of the run-time scheduler. Afterwards, they are updated each time that a task is reused or is loaded at a RU. Figure 48 presents the pseudo-code of the replacement heuristic.

To generate the Replacement list, this module identifies which tasks can be reused, and for each potentially reusable task it adds a node to the Replacement list. These nodes are sorted according to the run-time schedule. Thus, they are sorted following their execution order. The Non-reusable unit list is initialised with a node for each RU that cannot be reused during the current period. Finally, the Temp list is set to NULL. This list is used to prevent that two initial tasks of the same task-graph could be assigned to the same physical unit.

If all the tasks are executed just once each period, the number of nodes in the replacement plus the non-reusable list will be the same as the number of RUs. Nevertheless, if a task is executed more than once in the current scheduling, a node is added to the Replacement list for each execution of this reusable task. To control this recurrence a counter is associated to each RU. This counter is initialised with the number of times that the current task assigned to the RU is executed. With this counter, it also stores the moment when the task is executed for the first time. This information is needed to correctly apply the LFD strategy, since it is necessary to predict when the task-graph will be scheduled in the next iteration.

Once the lists have been initialised the reuse and replacement modules use these lists to identify the virtual units with the physical units. This process is carried out sequentially for each task-graph, attempting to find the identification that maximises the reuse.

```
# Initialization module:
inicializate (&replacement_list, &initial_scheduling);
inicializate (&non-reusable_list, &initial_scheduling);
inicializate (&temporal_list, empty_list);
for (i=0; i<number_of_task-graphs; i++)
{
        # Reuse module:
        identify_reusable_tasks (&replacement_list,
                &initial_scheduling, &reusable_tasks,
                        &initial_tasks_to_be_loaded);

        # Replacement module:
        move_nodes (&reusable_tasks, &replacement_list, &temporal_list);
        for (j=0 ; j<initial_tasks_to_be_loaded; j++)
        {
                choose_target_unit (&replacement_list,
                        &non-reuseble_list, &target_node);
                move_node (&target_node, &temporal_list);
        }
        empty_temporal (&temporal_list, &replacement_list);
}
```

Figure 48. LFD replacement heuristic pseudo-code.

First, the reuse module looks for reusable tasks reading the Replacement list (*identify_reusable_tasks*). To find them, it does not need to read the whole list because it is managed in such a way that at each moment the first nodes correspond to the current task-graph. Hence, when a module reads a node from another task-graph, the search finishes.

Each time that a reusable task is identified, the corresponding node is moved from the Replacement list to the Temp list. In the pseudo-code this process is carried out by the *move_nodes* function using the reusable task list obtained in the previous step.

The next step consist in identify the remaining virtual units. To do this for each initial task to be loaded, the *choose_target_unit* function is called in order to select one of the physical units available. Applying the replacement heuristic this is done in a very straightforward way: if the *non-reusable_list* is not empty one of its nodes is selected; otherwise, the last node from the *replacement_list* is selected. Since these lists have been sorted following the execution order of the schedule selected by the run-time scheduler, the last node corresponds to the reusable task that has been scheduled for being executed, or that is expected to be executed, further from the current point of time.

When several nodes associated to the same task exist, the scheduler only takes into account the first one, since it is the one that represents the closest execution. The remaining nodes associated to the task continue invisible until the first node disappears. This guarantees the correct application of the LFD strategy.

After selecting the target node, the *move_node* removes it from the Replacement list and adds it to Temp with the information of the current task loaded.

Once all the virtual units have been identified with physical ones, the Temp list is emptied (*empty_temporal*). The assignment of these nodes depends on whether they have been reused or loaded, and if the corresponding task appears one or several times in the schedule (this information is stored in a counter in each RU).

The possible cases are:

Case 1: The node of Temp corresponds to a reused task that appears multiple times in the scheduling. In this case the node is removed and the counter associated

with that unit is decremented by one. The remaining nodes associated to the same task remain in the Replacement list. These nodes were invisible to guarantee that the LFD heuristic works correctly. Now the first of these nodes starts to become visible.

Case 2: The node of Temp corresponds to a reused task that appears only once in the schedule. In this case the node is inserted again into the Replacement list. The nodes are inserted at the end sorted according to the start of their execution.

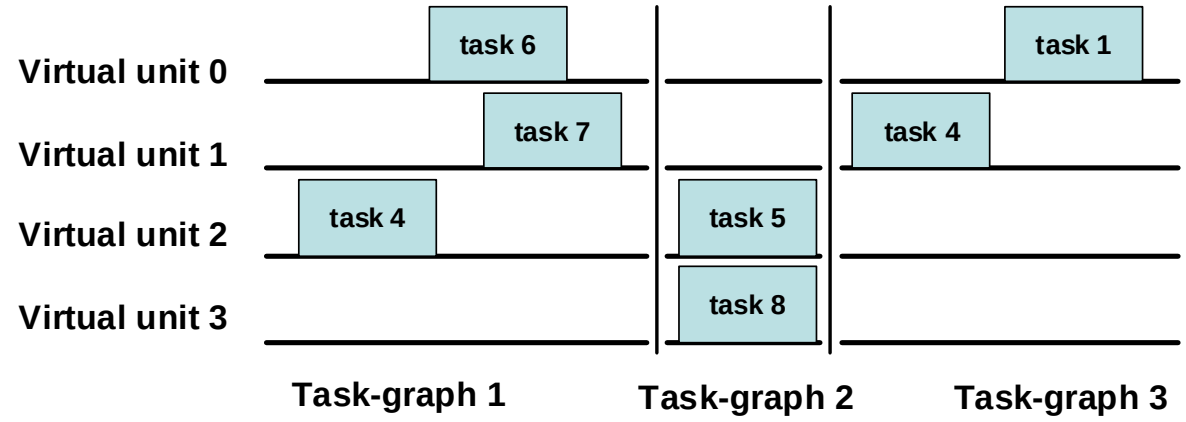
Case 3: The node of Temp corresponds to a task that has just been loaded. As in the previous case, the node is inserted at the end of the list. When in the same task-graph several tasks are loaded in the same unit, only the last of them is stored, since it is the only one that remains loaded when the identification process of the virtual units of the next task-graph starts.

After removing the nodes from the Temp list, the Replacement list holds the nodes of categories one and two. Since the nodes of the Temp list are stored at the end of the Replacement list, the nodes from the category one (those that have never been in the Temp list) are at the beginning of the list, meanwhile the nodes of the category two are at the end. Hence, to select the lowest priority node, the heuristic only has to select the last node of the list.

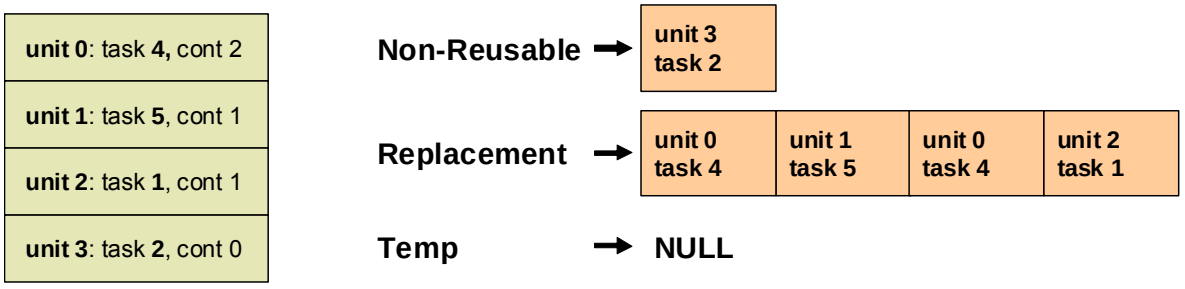
The nodes of the second category are stored at the end of the Replacement list taking into account when they were executed the first time. Assuming that the next selected schedule is going to be the same as the present one, the last node of the Replacement list would hold the tasks that would be executed further from the current

point of time. The next figures depict how the replacement heuristic works using a step by step example (from Figure 49 to Figure 58)

Figure 49 presents the initial state and the selected scheduling for the task-graph scheduler.



a)



b)

Figure 49. Example of the replacement list management. a) Initial assignation and schedule selected by the run-time scheduler. b) Lists generated by the initialization module.

In this example the input is the schedule of three task-graphs. The initialization module parses this schedule and initializes the three lists finding that tasks 4, 5 and 1 may be reused in this iteration. In addition, it detects that task 4 can be reused twice. Hence, it generates four nodes for the Replacement list (one for tasks 1 and 5 and

two for tasks 4). In the Free list only the node 3 is inserted which corresponds with the task 2. The initialization module also assigns the initial values of the counters, assigning a value of 2 to the counter of the unit 0 since the task 4 is going to be executed twice during the current iteration. The two nodes with Task 4 also include the information of when the first time of the task was scheduled, since it will be needed to correctly apply the heuristic.

Step 1:
Task-graph 1
Task 4 is reused, virtual unit 2 is identified with physical unit 0

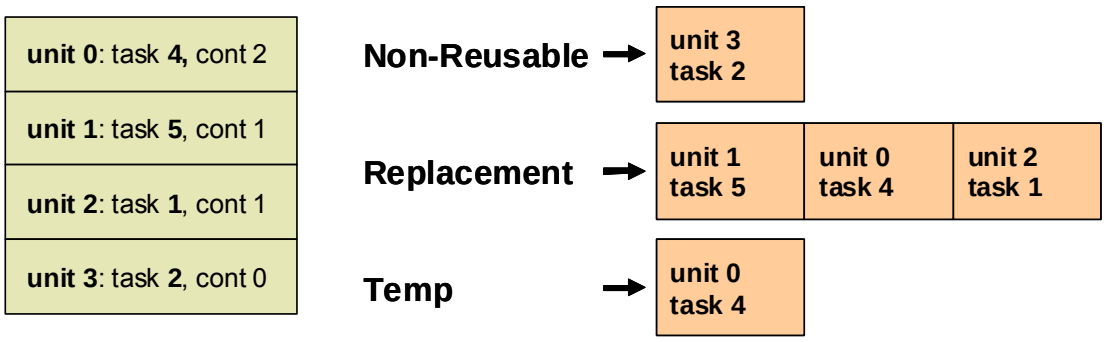


Figure 50. Example of the replacement list management. Step 1: Task 4 management.

Step 1: The reuse module detects that task 4 can be reused. To this end virtual unit 2 is identified with physical unit 0. Thereafter, the first node corresponding to task 4 is removed from the Replacement list and added to the Temp one.

Step 2: The task 6 is not loaded, so it must be loaded in one of the existing units. Hence it has to replace one of the previously loaded tasks. In this case the replacement module selects the unit 3, overwriting Task 2, because it holds a task from the non-reusable list with the minimum priority. Afterwards, the node that

corresponds to task 2 is deleted, and a new node with the information of task 6 is stored in the Temp list.

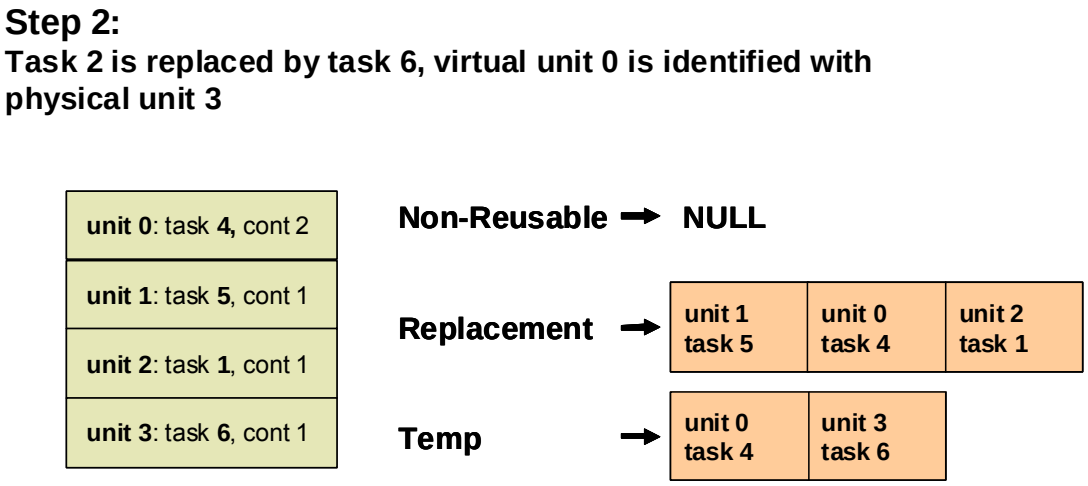


Figure 51. Example of the replacement list management. Step 2: Task 6 management.

Step 3: Task 7 is not loaded and it must be loaded in one of the existing units. This time the non-reusable list is empty. Hence, the task to replace must be selected from the Replacement list. In this case according to the LFD replacement strategy task 1 is selected, since from all the tasks stored in the Replacement list, it is the one that is going to be executed further. Afterwards, the node that corresponds to task 1 is deleted from the Replacement list, and a new node with the information of task 7 is stored in Temp.

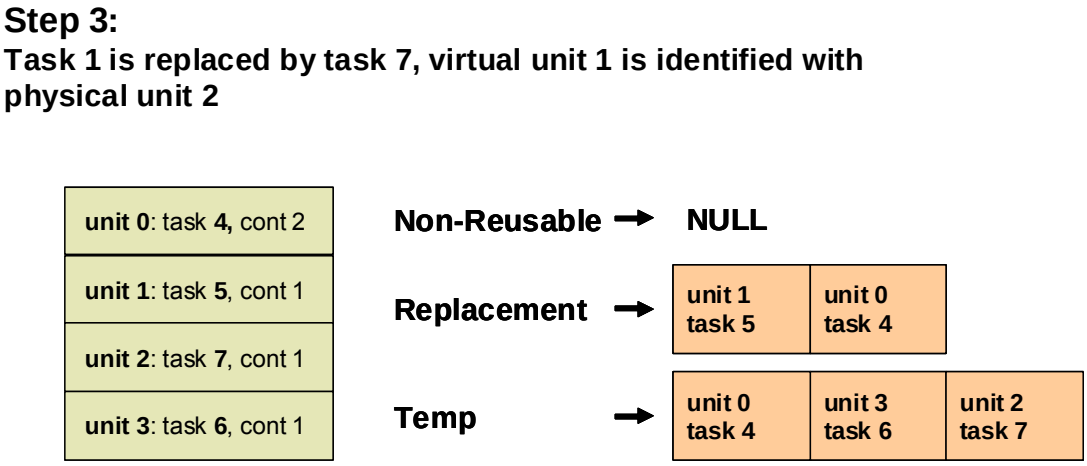


Figure 52. Example of the replacement list management. Step 3: Task 7 management.

Step 4: Once all the virtual units of the task-graph 1 have been identified, the process continues with the following task-graph, but previously all the nodes from Temp list are removed and stored in the Replacement list. Since already a node in this list corresponds to task 4, it is not needed to store another one, but only to decrement by one the counter assigned to this task unit.

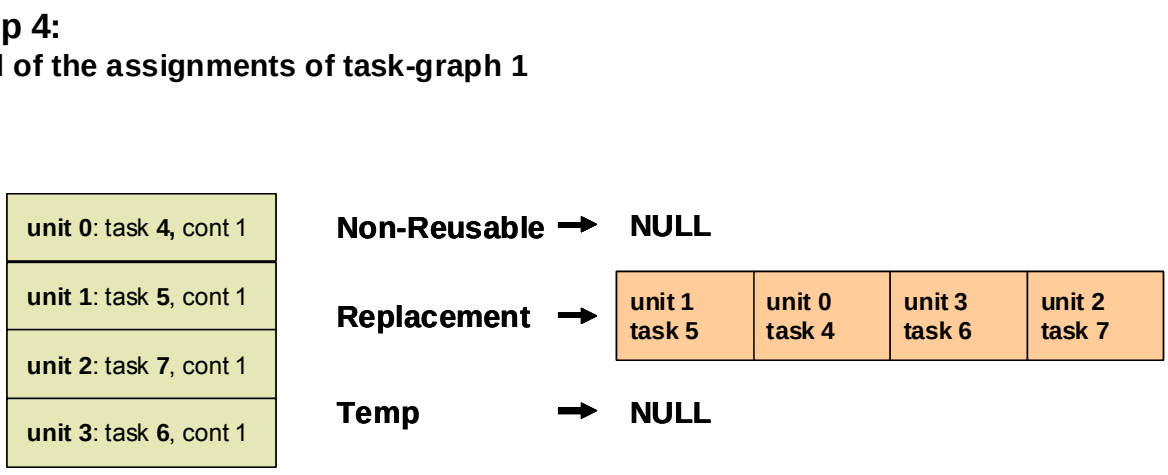


Figure 53. Example of the replacement list management. Step 4: Task 1 management.

Step 5: The task 5 of the task-graph 2 is analysed.

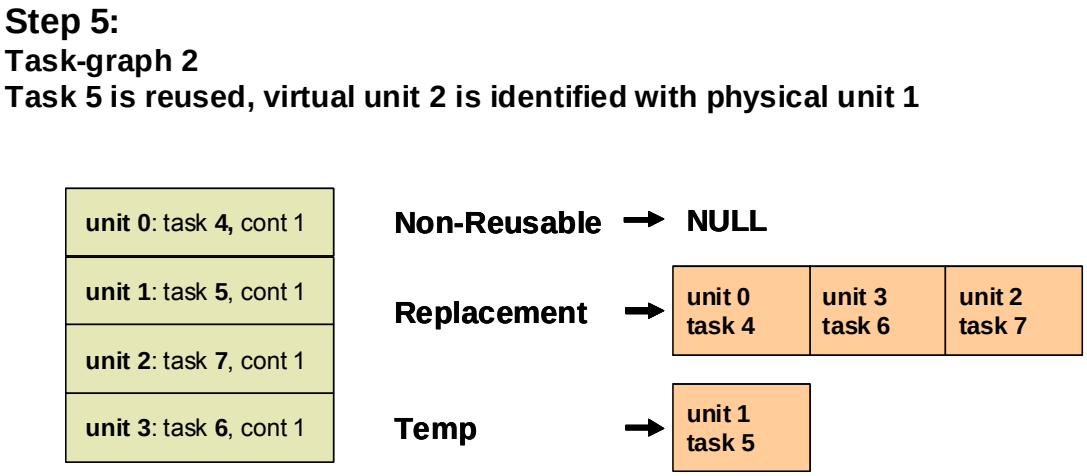


Figure 54. Example of the replacement list management. Step 5: Task 5 management.

Step 6: Task 8 is not loaded. As in Step 3 the task to replace must be selected from the Replacement list. As always, the last node from the list is selected. In this case it corresponds to task 7. It is interesting to remark how those nodes that have already been reused, or that have been loaded during this iteration (category 2 in figure 42) have always less priority than those nodes that can still be reused during the current iteration. Among the nodes from category 2 the task 7 is selected because it starts its execution later than task 6 and the nodes from category 2 are stored following their execution order during the current iteration.

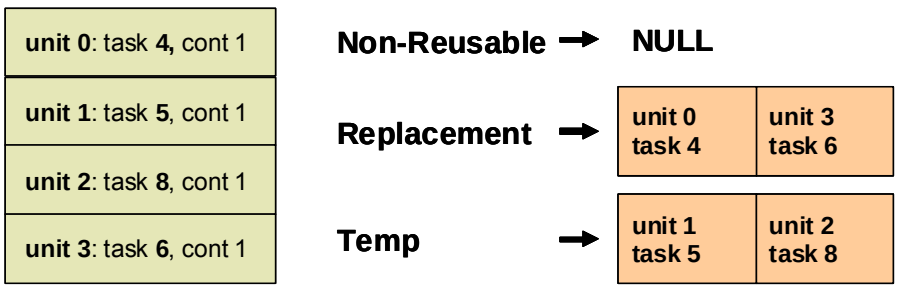


Figure 55. Example of the replacement list management. Step 6: Task 8 management.

Step 7: Similar to step 4.

Step 7:
End of the assignments of task-graph 2

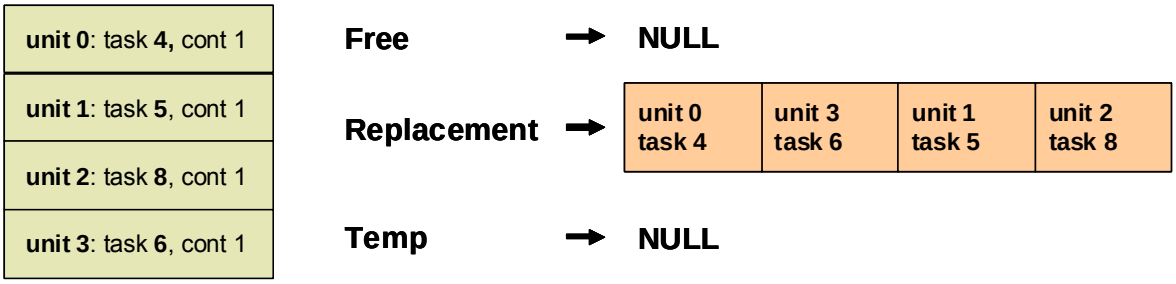


Figure 56. Example of the replacement list management. Step 7: End of the task 2 assigning process.

Step 8: Similar to step 1.

Step 8:
Task-graph 3
Task 4 is reused, virtual unit 1 is identified with physical unit 0

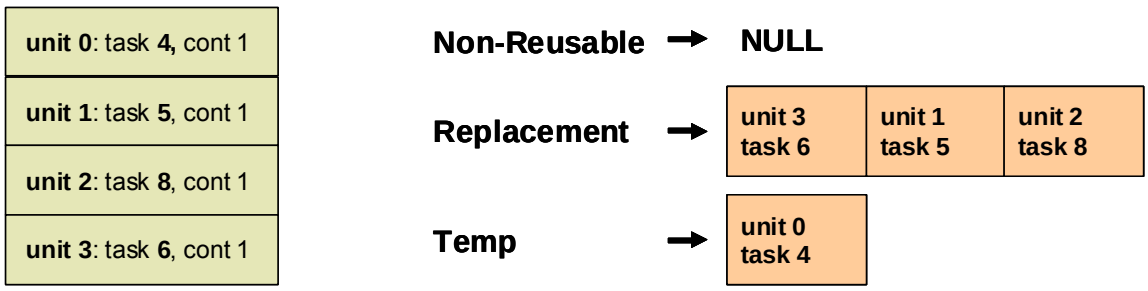


Figure 57. Example of the replacement list management. Step 8: Task 4 management.

Step 9: End of the assignment process. Task 1 replaces Task 8. Virtual unit 0 is identified with physical unit 2.

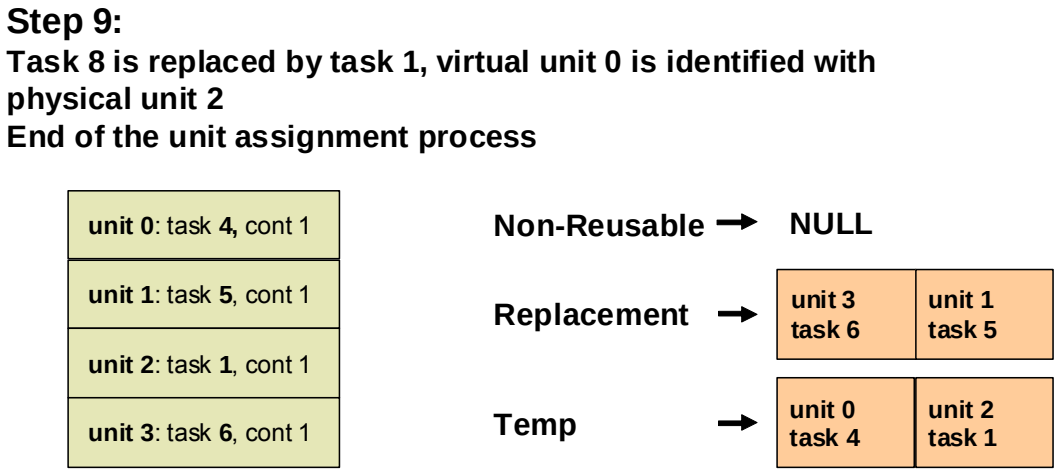


Figure 58. Example of the replacement list management. Step 9: Task 1 management and end of the identification process of the virtual units with the physical units.

Table 1 depicts the final assignment. As can be seen, the assignment process is carried out independently for each task-graph.

Physical RUs	*Task 1*	*Task 2*	*Task 3*
0	Virtual Unit 2		Virtual Unit 1
1		Virtual Unit 2	
2	Virtual Unit 1	Virtual Unit 3	Virtual Unit 0
3	Virtual Unit 0		

Table 1. Final assignment of the virtual units of task-graph 1, 2 and 3 (from figure 44) to physical units.

For this iteration of the example the result obtained is optimal taking into account the initial state of the RUs. Since, it is reused the only task that can be reused.

The complexity of the different operations of this assignment process is the following:

- Parsing the initial schedule and initialisation of the replacement and non-reusable unit list: O(*N**LOG(*N*)). Where *N* is the number of initial tasks assigned to the reconfigurable resources in the initial schedule.
- Updating the Replacement list each time that finish the assignment of all the virtual units of a given task-graph: O(*M*). Where *M* is the number of initial tasks assigned to virtual units of the given task-graph.
- Selecting the task that is going to be replaced: O(1).

The initialisation process is the most computationally intensive part. But it is still very affordable and it is just executed once during the assignment process of a sequence of task-graphs.

5.5. Experimental Results

Before analysing the results obtained applying the modules presented in this chapter, it is necessary to evaluate if it is possible to apply them at run-time, without generating important run-time penalties.

These modules have been designed taking into account that they have to apply efficient heuristics that obtain good results as fast as possible. Analysing these heuristics at run-time in a 200 MHz microprocessors, only the prefetch heuristic introduces a significant delay. For example, the prefetch heuristic lasts 4 μs to schedule the load of 13 tasks in the RUs. If it is necessary to schedule 20 task-graphs with these features, the whole process could be finished in 0.08 ms. Finally, taking

into account the time need to execute the replacement, reuse modules and to carry out the initialization, the global time needed to schedule the load of these 20 task-graphs would be less than 0.1 ms. This is an affordable penalisation, especially if we take into account that by applying our optimisation techniques to 260 loads (13*20) will probably achieve reductions in the execution time that widely compensate this small penalty. As an example we are going to consider that the time needed to load a task into a RU (reconfiguration latency) is 4 ms that is the minimum time needed to reconfigure a tenth of a FPGA Virtex XC2V6000. Thus, the execution of these modules would be clearly favourable if they could hide the time needed to load one of the 260 tasks. Hence, it would save forty times more time that the time invested in carrying out the calculations.

5.5.1. Experiments for a set of multimedia applications

Now that we have proven that this module can be applied at run-time without generating too much delay, we are going to test it with four multimedia applications. The features of these applications and their execution time without taking into account the reconfiguration latency (**Ideal Tex**) are presented in Table 2. The first application is a pattern recognition application that applies the Hough transformation [Leig92] over the input image in order to identify geometric patterns. The two next applications are two different versions of the JPEG decoder [JPEG]. The first version decodes the image block by block sequentially, while the second one can work with

three blocks in parallel. Finally, the forth application is a video encoder that follows the MPEG standard [MPEG].

Task-graph	*Task Number*	*Ideal Tex*
Patters Rec. (Hough)	6	94 ms
JPEG dec.	4	81 ms
Parallel JPEG dec.	8	57 ms
MPEG enc.	5	33 ms

Table 2. Set of multimedia task-graphs.

For the MPEG encoder the results of Table 2 are averages values, since both the execution time and the penalty depend on the kind of image to be encoded (three kind of images, kind B, kind P and kind I are taken into account by the standard [MPEG]).

Table 3 presents the reconfiguration overhead for these applications assuming that the reconfiguration latency is 4 ms. In Table 3 **Ideal Tex** is the execution time when no reconfiguration is needed, just as in the previous table. **Penalty** is the execution-time increment when all the tasks must be loaded. This data have been calculated assuming that all the tasks have to be loaded and no prefetch heuristic has been performed, that is, the system attempts to load a task when its execution must start. Finally, **Prefetch** is the execution-time increment for the same case than Penalty but using the prefetch module. In these examples, applying the prefetch module to schedule the reconfiguration leads to large reconfiguration overhead reductions (approximately by a factor of 4).

Task-graph	*Ideal Tex*	*Penalty*	*Prefetch*
Patters Rec.	94 ms	+17%	+4%
JPEG dec.	81 ms	+20%	+5%
Parallel JPEG dec.	57 ms	+35%	+7%
MPEG enc.	33 ms	+56%	+18%
Total	**265 ms**	**+26%**	**+7%**

Table 3. Effect of the reconfiguration in a set of multimedia task-graphs.

As it can be seen in the last row, if we execute these four applications sequentially without including any specific support for the reconfigurations, the penalty due to the reconfiguration latency entails on average a 26% increment of the execution time. This overhead is reduced to 7% when including the prefetch module, that is, the prefetch module removes 73% of the initial penalty. The data of this table have been obtained assuming that all the tasks are executed in a RU and that none of them is reused. Nevertheless, if the reuse module is also added, the reusable tasks can be identified at run-time, reducing even more the global penalty.

In order to evaluate the effect of including the reuse module together with the prefetch one, we have simulated the execution of 1.000 iterations of this set of task-graphs. Since our modules have been designed to deal with applications with dynamic behaviour, at the beginning of each iteration the task-graphs to be executed are selected randomly. Hence, each iteration can be different to the previous one. Figures 59 and 60 depict the result of this simulation. The first figure depicts the percentage of reused tasks for different number of RUs, whereas the second figure

depicts the final penalty. As it was expected, the percentage of tasks reused increases linearly with the number of units. Hence, the penalty due to the reconfiguration processes decreases linearly as well.

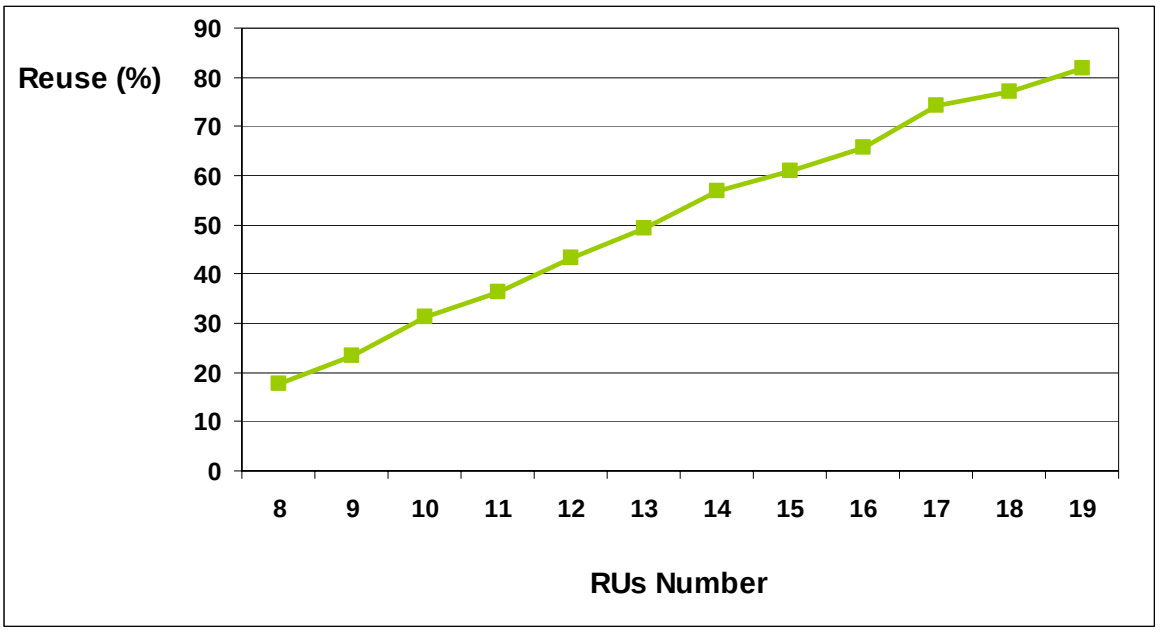


Figure 59. Percentage of reused tasks for different number of reconfigurable units. Results obtained for the set of multimedia application presented in table 2.

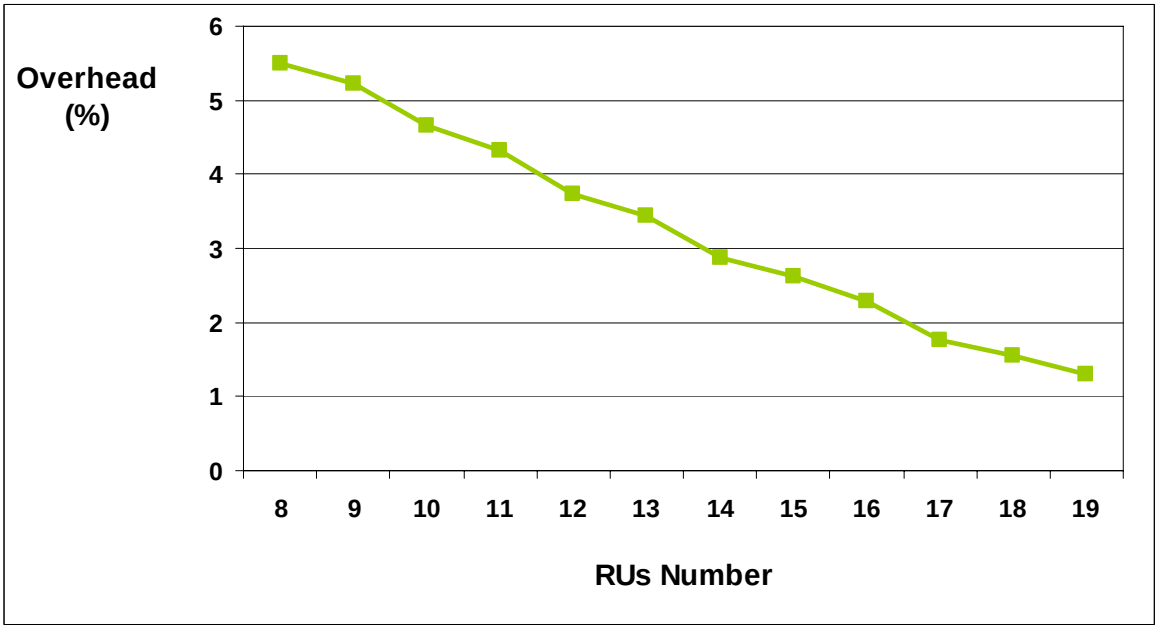


Figure 60. Penalty due to the load of non-reused tasks. Results obtained for the set of multimedia application presented in table 2.

As it can be seen in these figures, important penalty reductions are achieved even for a small percentage of reused tasks. For example, for 8 reconfigurable units, with a reuse percentage lower than 20% (since 23 tasks are sharing 8 RUs) the reconfiguration penalty falls around 30%, standing at 5%. With a high reuse percentage (80% for 19 RUs) the reconfiguration penalty stands at 1%. When enough task repetition is present, the reuse module allows improving easily the results obtained by the prefetch module, without complex computations. This module has a passive behaviour and its unique activity consists of identifying the virtual units with the physical ones to increase the tasks reused. When a task that cannot be reused has to be loaded in a RU, this RU is randomly selected without taking into account the result of this decision. This passive behaviour can be improved by activating the replacement module. Thus, in order to increase the percentage of reused tasks, the scheduler uses a replacement heuristic that takes into account the effects of its decisions. In figures 61 and 62 the results obtained activating the replacement module are compared with those obtained previously (only using the reuse module).

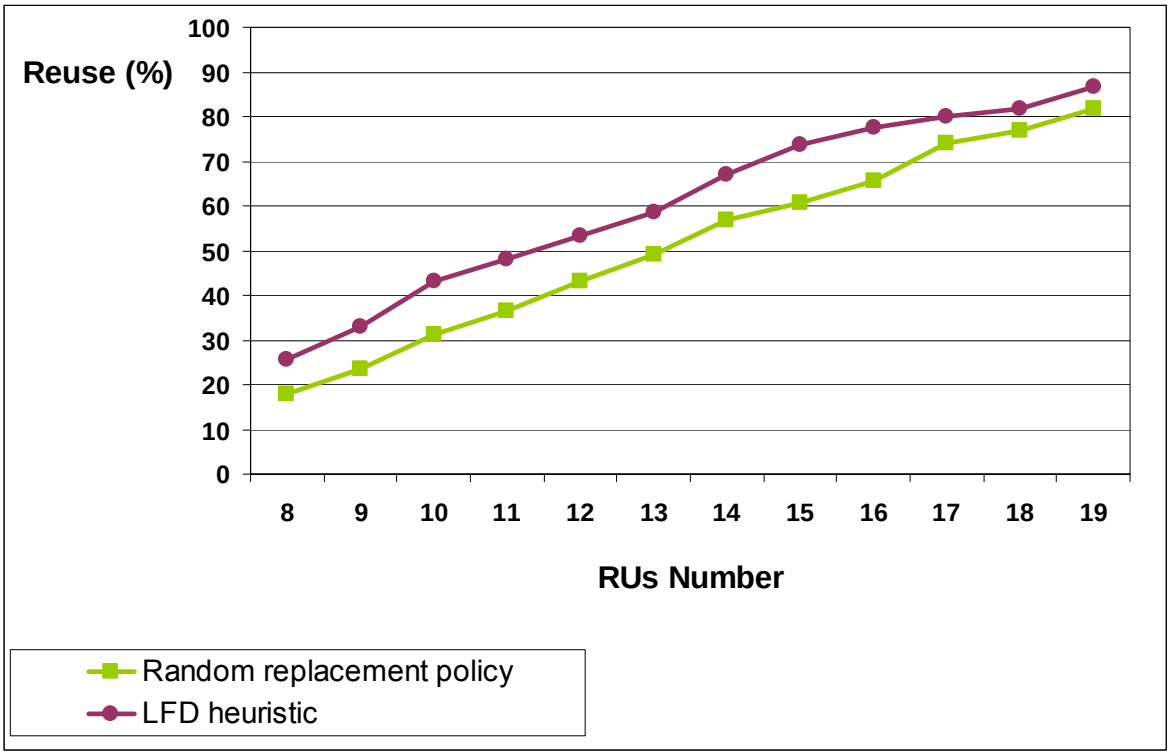


Figure 61. Comparison between the reused task percentage for different number of RUs with and without the replacement module. Results obtained for a set of multimedia application.

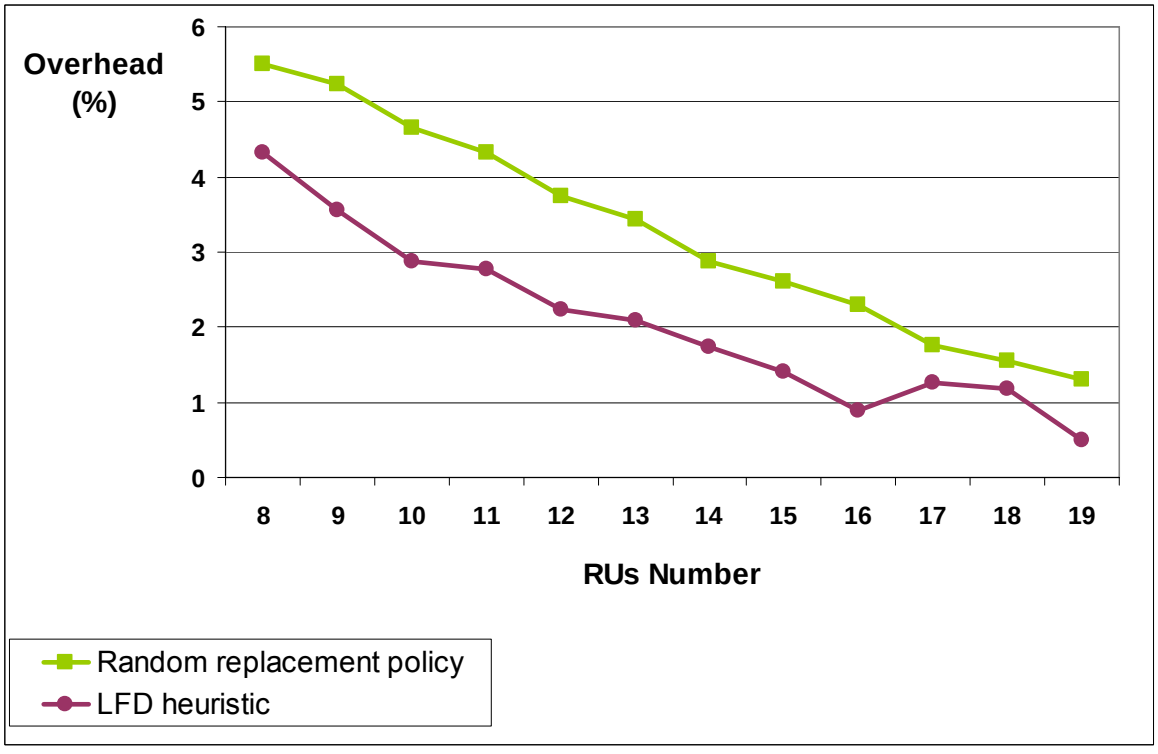


Figure 62. Comparison between the tasks load penalty for different number of RUs with and without the replacement module. Results obtained for a set of multimedia application.

As it can be seen in these figures, the replacement module introduces substantial improvements with regard to applying the pseudo-random replacement strategy, increasing the reused task percentage with the resulting penalty reduction. For example, for 8 RUs the penalty is reduced by 27% and for 16 RUs by 56%. Only when the number of units is big, both options provide similar results. Table 4 presents the most significant results obtained for this set of applications. As it can be seen in this table, the most important penalty reduction is due to the addition of the prefetch module that it is able to reduce it in a factor of 4. The reuse and replacement modules does not achieve such an important reduction, but they are also able to reduce considerably the penalty (with 10 units both modules remove 40% of the remaining penalty)

Initial Penalization	+26%
Penalization with the prefetch	+7%
Penalization with the prefetch and reuse modules (10 units)	+4.7%
Penalization with the prefetch, reuse and replacement modules (10 units)	+2.8%

Table 4. First experiment results summary.

A surprising result can be seen in Figure 62. Comparing the results obtained for 16, 17, and 18 RUs, it can be seen that the best result is it obtained for 16 RUs. Taking into account that increasing the number of RUs, also increases the reused task percentage. It seems illogical that increasing the reuse also increases the penalty. However, this behaviour is possible.

In particular, if the load of every task generates the same penalty, any increase in the percentage of reused task would lead to a penalty reduction. Hence, this anomalous behaviour would never appear. Nevertheless, in our system the penalty generated by a task load depends on up to what point the reconfiguration latency of this task can be hidden by the prefetch module. Hence, for some tasks this latency would be completely hidden, whereas for others all this latency, or at least part of it, will generate a penalization. Since the penalty introduced by a task load is not constant, it is possible that a replacement policy that reuses n tasks will be worse than a replacement policy that reuses a different subset of n-1 tasks, although most times more reuse will lead to better results.

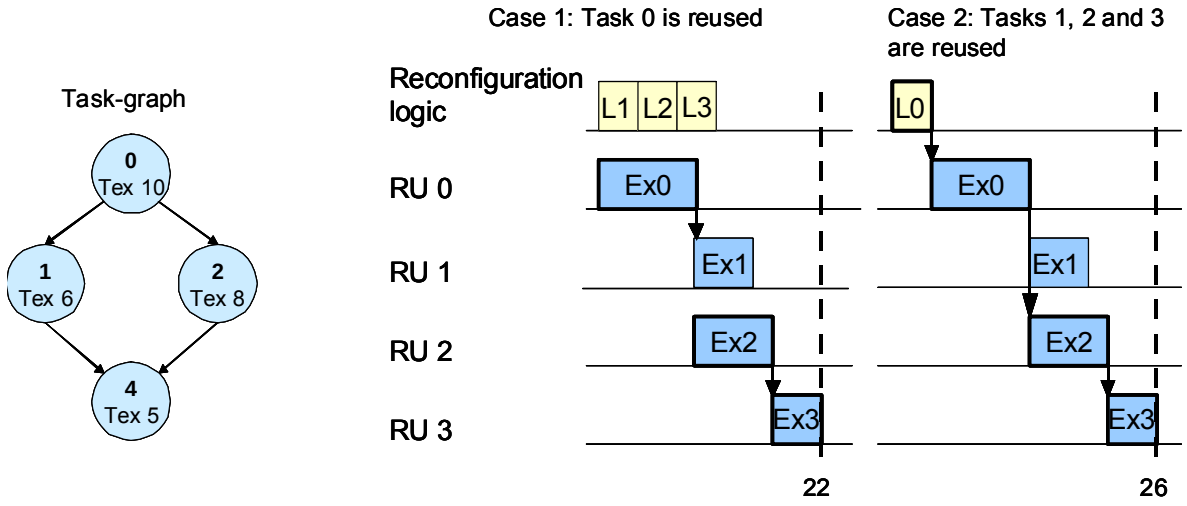


Figure 63. Comparison between reusing only task 0 and reusing all the tasks but task 0 for the task-graph of the figure and for four RUs. Tex: Task execution time. Ex i: Task i execution. L i: Task i load. Arrows show the dependences into the critical path.

Figure 63 depicts a clear example about how the increment of reused tasks not always entails a reduction of the execution-time. In the case 1 of the figure only one task is reused but the penalty due to the other tasks is zero, since none of them delay

the critical path of the task-graph. The opposite case is presented in Case 2, where three tasks are reused, but the only task that has to be loaded implies a 4 ms overhead in the whole system execution. This is due to the fact that the prefetch module cannot hide the time needed to load this task, whereas it is able to hide the load of the other three.

5.5.2. Recent Advances

Although the results of the modules presented in this chapter are very satisfactory, in the recent years some improvements have been carried out in order to further improve the results and to reduce the computations carried out at design-time. A detailed explanation of these improvements is out of the scope of this thesis but I would like to provide some basic hints with the proper references:

- Prefetch heuristic: we have developed a new one where almost all computations are carried out design-time. The idea is that an initial reconfiguration schedule is carried out at design-time under certain assumptions, and at run-time this schedule is slightly modified to fit the actual run-time situation. This approach achieves run-time flexibility while almost no additional computations are carried out at run-time. In [RMCC08] we compare this approach with the previous one described in the thesis, and the results demonstrate that the prefetch schedules are almost similar, whereas the run-time computations are reduced 20 times.

- Replacement heuristic: We have slightly modified the replacement module to prevent the anomalous results described in Figure 63, and at the same time achieve further reductions in the reconfiguration overhead. To this end we analyse at design-time the task graphs in order to identify which tasks cannot be prefetched without introducing any delay. Once these tasks are identified the replacement module assigns them more priority than to the remaining tasks. This approach leads to significant reductions of the reconfiguration overhead while introducing almost no complexity in the replacement module. Further details are described in [RVMC04].

- Hardware support: in this thesis we have executed all this modules in an embedded processor. However, in parallel some colleagues have been developing a hardware micro-architecture with HW support to apply these techniques. With this approach it is possible to deal with the task-graphs at run-time, applying all these optimization techniques, while introducing a delay of only a few clock cycles due to the run-time computations. Some initial results of this project can be found in [CGGR07] [CGRM08].

Chapter 6:

A Configuration Memory Hierarchy to Optimize the Scheduling Process

As we have mentioned in the previous chapters, run-time reconfigurable resources present many of the features such as high performance, flexibility and reusability demanded by next generation embedded systems. In addition, many emerging reconfigurable architectures have been optimised for low power. However, carrying out run-time reconfigurations often involves a costly reconfiguration overhead both in execution time and in energy consumption. The work presented in the previous chapter only tackles the reconfiguration execution-time overhead, regarding the extra energy consumption due to the reconfiguration. In this chapter, we present a new complementary approach that significantly extends the already presented work, while attempting to reduce the reconfiguration energy overhead as

well. To this end, we propose a configuration memory hierarchy, with a shared memory layer consisting of a module optimised for performance combined with a module optimised for energy-efficient accesses. For this hierarchy, we have developed a mapping algorithm that decides where to load each configuration in order to achieve significant energy savings without introducing any performance degradation. This approach could, in principle, be applied also to the instruction memory organisation of an instruction-set programmable processor, with some adaptations.

6.1. Higher Performance and Low Energy for Dynamically Reconfigurable HW

As we have mentioned before, embedded system design complexity is rapidly increasing as applications are becoming more and more demanding and the users are expecting higher performance and extended battery life. Hence, designers must optimise their systems both for performance and for reduced energy consumption. However, these two objectives frequently drive the design process in different directions. Thus, those optimisations that improve the system performance often lead to energy penalisations, whereas those that reduce the energy consumption frequently lead to performance degradation. Reconfigurable HW is used to implement hardware accelerators that take advantage of the inherent parallelism of each task. In addition, they provide interesting features such as flexibility and reusability. Moreover,

many emerging coarse-grain reconfigurable architectures have been designed for low-power and low-energy (LE) consumption computations.

As we have already explained one of the main contributions of reconfigurable resources over ASICs is the run-time flexibility they provide. Thus, new configurations can be loaded at run-time in order to adapt the system to new situations. This feature is especially interesting to deal with current multimedia applications, such as digital video and three-dimensional games, as they exhibit highly dynamic and non-deterministic run-time behaviour, as well as a very variable workload. However, in order to efficiently tackle this dynamism, very frequent reconfigurations are demanded, and these reconfigurations will have a clear impact on both performance and energy consumption. For example, current multimedia standards, such as MPEG-4 [KnSM99], support digital video and three-dimensional game applications, where the number and type of the objects to decode and visualise can change at run-time from one frame to another. Moreover, to reach the standard quality level, the system must be able to decode and visualise a frame in less than 33 ms. Hence, if the decoding computations are executed in reconfigurable resources, it is likely that the system will need to carry out reconfigurations every few milliseconds in order to adapt itself to all the variations in the number and type of the objects to decode. Clearly, carrying out these reconfigurations with conventional solutions will generate some delays in the system execution. Moreover, loading a new configuration involves a large amount of read/write operations with the corresponding energy penalty.

Most of the research groups presented in chapter 3 are mainly focused in the minimisation of the reconfiguration overhead. However, many researches have

pointed out that, in embedded systems, the energy consumption due to the instruction memory hierarchy still represents a very important percentage (30%) of the overall energy consumption [BBCS01], [JBVC05]. This is also true for fine-grain [ShJh02] and coarse-grain [BJVL03] reconfigurable architectures, as long as frequent reconfigurations are demanded. Hence, reconfigurable systems with energy-efficient reconfigurations are strongly desired. However, until now very few efforts have been carried out on this subject, not only from the academic research groups but also from the industry.

In this chapter we present a High-Speed/Low-Energy (HS/LE) configuration memory hierarchy that can be used to provide fast reconfigurations when they are especially critical for the system performance and, at the same time, to reduce significantly the average energy overhead. This hierarchy has been developed for systems dealing with highly dynamic applications where reconfigurations are carried out very frequently (every few milliseconds). In addition, we have developed a configuration mapping algorithm that takes advantage of the special features of this configuration memory hierarchy. Finally, we have integrated this mapping algorithm into an existing reconfiguration manager presented in chapter 5 and we have tested it with a set of representative multimedia applications.

6.2. Configuration Memory Hierarchy

The typical configuration memory hierarchy (Figure 64) for a reconfigurable hardware platform is composed of a reconfigurable fabric that stores the

configurations that are ready for being executed and an off-chip memory where the remaining configurations are stored. These configurations can be loaded from the external memory using a dedicated reconfiguration circuitry. This scheme is usually present on fine-grain architectures, as FPGAs, and it is the configuration memory hierarchy that many researchers have targeted.

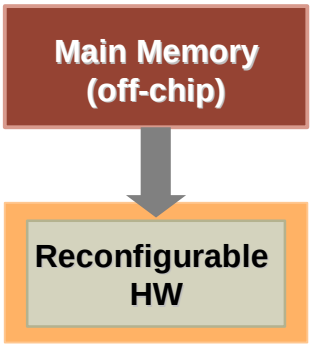


Figure 64. Typical memory hierarchy for fine-grain platforms.

One interesting improvement often introduced for coarse-grain devices (and, sometimes, also for fine-grain devices [BJKM03]) consists in adding a smaller intermediate on-chip configuration memory, where the configurations of the running tasks are stored. The scheme of this improvement is depicted in Figure 65. This memory is sometimes called configuration cache, although normally it is not a real cache memory but a static random access memory (SRAM) controlled by SW (i.e. a scratchpad). The configuration memory hierarchy is critical for the system, not only for the heavy configuration traffic required by dynamic application execution, but also for its energy consumption. Including this on-chip memory level may drastically increase the energy efficiency of the configuration memory hierarchy because the on-chip memory is smaller and no need exists any longer to continuously access a high-

capacitance off-chip bus [FPCK97]. This internal configuration scratchpad memory is usually a high-speed (HS) SRAM. SRAMs typically have high performance ratios per price unit. In addition, owing to the new development techniques applied, its cost is, at present, affordable. However, despite the fact that several improved LE techniques have been applied to these HS SRAMs [MaCC01], they still generate an important percentage of the total energy consumption of the embedded system.

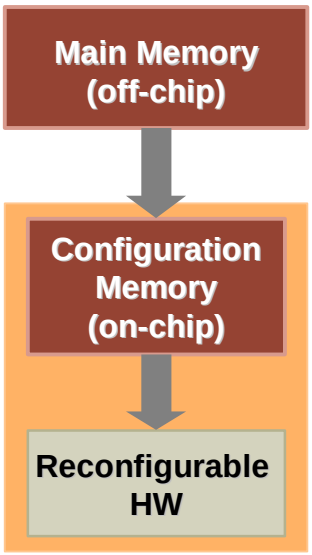


Figure 65. Typical memory hierarchy for coarse-grain reconfigurable HW.

In recent years, extensive efforts have been focused on reducing SRAM energy consumption. As a result, a new type of memory oriented to LE is currently available in the market with similar features to the HS ones, but with better energy efficiency at the cost of worse speed ratios. Different memory manufacturers have introduced some of these innovating techniques in the design of LE SRAMs. For example, Virage Logic [Vira09] and Micron Technology [Micr09] have presented memory modules optimised for power and energy efficiency, instead of performance. Moreover, CACTI [CACT08], a well-known simulation tool for cache memories, has

introduced in its latest versions different technology models that can be used to simulate either HS or LE SRAMs. The following figures depict an example of the available energy/delay trade-off obtained using CACTI. They include the data regarding the delay and the dynamic energy consumption per read port for memory modules of different sizes (1 Kbytes, 2 Kbytes, 4 Kbytes, 8 Kbytes, 16 Kbytes, and 32 Kbytes). On the one hand, Figure 66 depicts the access time in ns for the proposed memory sizes. On the other hand, Figure 67 depicts the total dynamic energy consumption per read port in nJ for the same cases.

We have obtained this numbers using the following parameters:

Parameter	
# banks	8
# read/write ports	1
# read ports	1
# write ports	1
# single ended read ports	1
# bits read out	8
Technology node (nm)	45nm
Temperature	320º F
Interconnect projection type	conservative
Type of wire outside mat	global

Table 5. CACTI used parameters

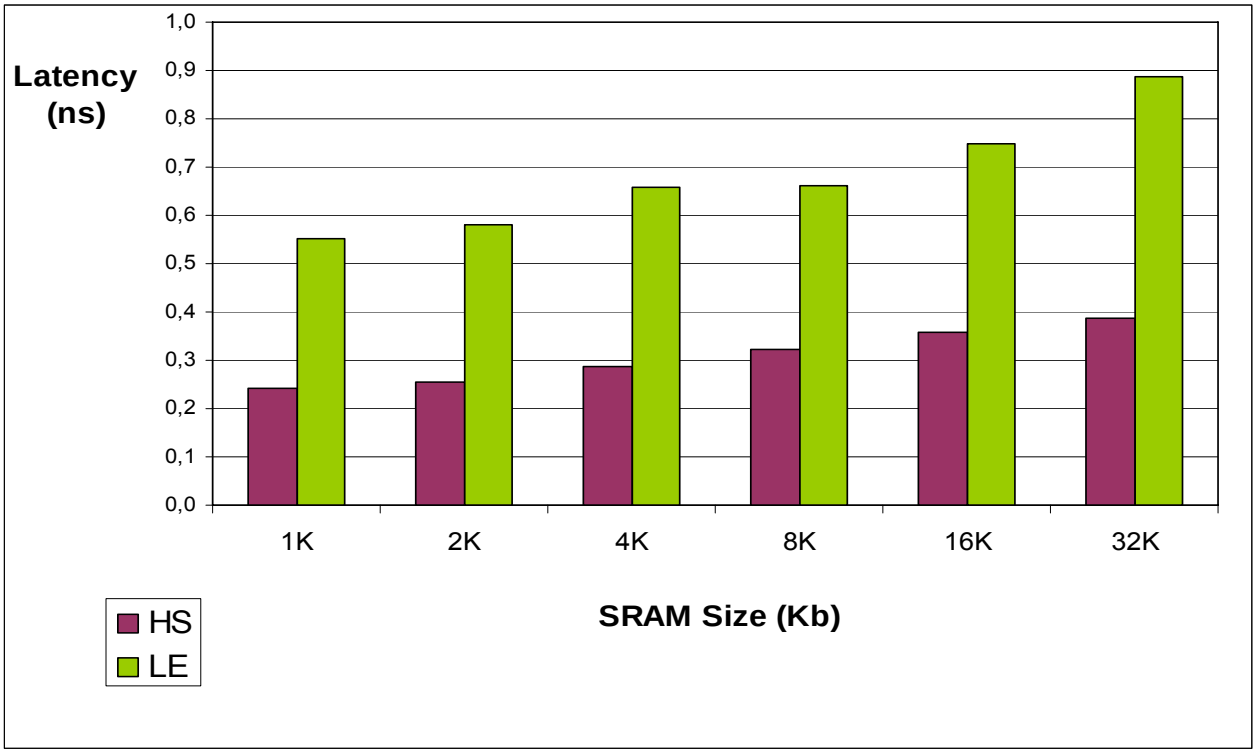


Figure 66. Access time for different memory sizes, using CACTI.

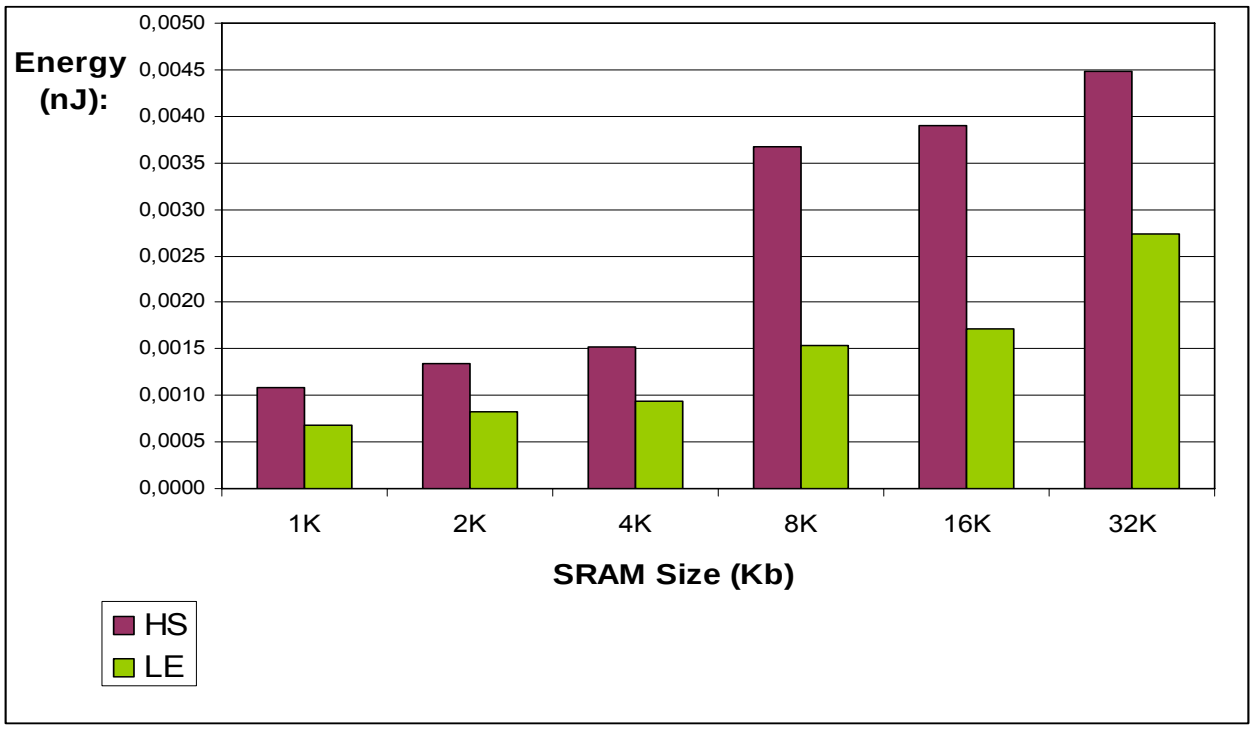


Figure 67. Total dynamic energy consumption per read port for different memory sizes, using CACTI.

As can be seen in these graphics a clear performance-energy consumption trade-off exists. Consequently, the embedded system designers must select the appropriate memory for their platform among the wide number of possibilities available in the market, that is, they should select the memory module that provides the best performance, or the one with less energy consumption per access, or something in the middle. However, selecting a memory optimised for high performance usually involves energy consumption overheads, whereas selecting a memory optimised for reducing the energy consumption may lead to important performance degradation since more data-path cycles are needed per access. As designers need both high performance and low energy consumption features, we propose to include at least two different types of memories in the configuration memory hierarchy: one optimised for HS and the other optimised for LE. Hence, we are potentially supplying high performance and low energy features to the configuration memory hierarchy. The goal of this scheme is to reduce the average energy consumption of the system, while maintaining high performance. Figure 68 depicts this configuration hierarchy memory scheme. Our approach presents a new challenge, because it is necessary to decide at run-time where to load each configuration for each part of the application sequence, in the HS memory or in the LE memory. Hence, we have developed a hybrid design-time/run-time configuration-memory mapping algorithm that takes these decisions automatically and we have integrated it into our previous reconfiguration manager.

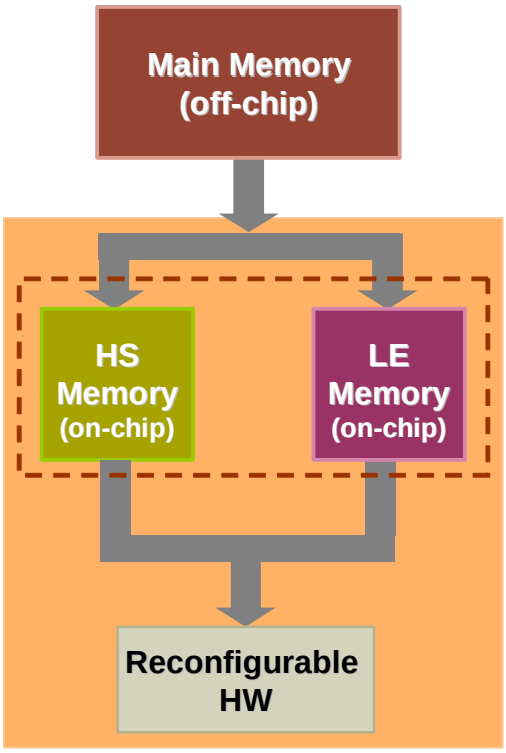


Figure 68. Proposed scheme of configuration memory hierarchy.

To introduce the idea of our HS/LE configuration memory hierarchy, in this chapter we assume that when the execution of a new application starts, the configurations of all the tasks of the task-graphs assigned to HW are preloaded to the scratchpads. This assumption should be acceptable for most of our target applications because only the most computing intensive kernels are typically assigned to the reconfigurable units (RUs). Under the 10–90 rule, generally 90% of the computations are carried out by 10% of the code. Hence, it should be possible to load the tasks for that 10% while starting the execution of the application. If this assumption is not correct, either only the most frequent tasks should be preloaded in the scratchpad or a replacement policy for the scratchpad should be included in the system. In the next chapter we present the extension of this approach to tackle our

HS/LE configuration memory hierarchy system when all the tasks assigned to HW cannot be preloaded into the on-chip memory layer of our configuration memory hierarchy.

6.3. Motivational Example

We have already illustrated how beneficial it can be to provide reconfigurable systems with a HS/LE configuration memory layer for dynamic applications execution with the motivational example in chapter 1. We will go back to that example in this section, in order to explain in detail our mapping algorithm. Figure 69 depicts the task-graph of this example.

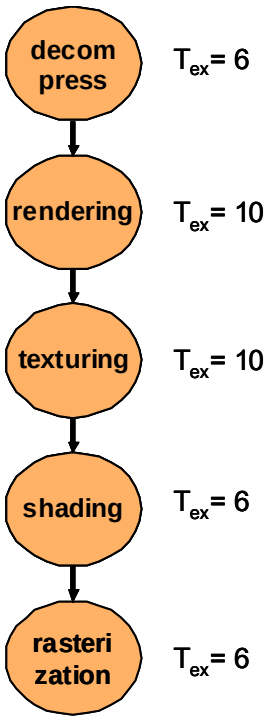


Figure 69. Tasks for displaying a 3-D object.

Figure 70 shows the schedule of this task-graph execution when only a HS dynamic random access memory (HS DRAM) is used to store the configurations of the running applications. As we have already explained, current reconfigurable systems have only one reconfigurable circuitry to carry out the reconfigurations of the different RUs. Therefore simultaneous reconfigurations are not supported.

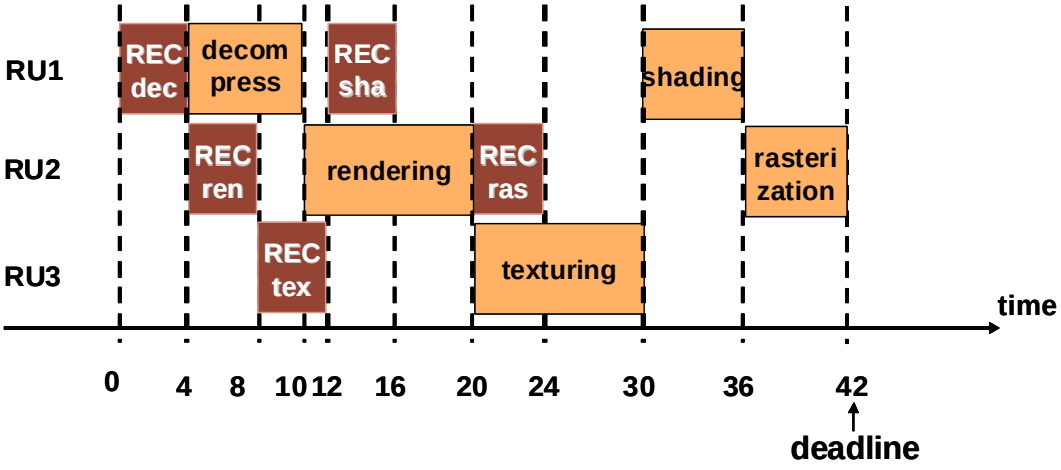


Figure 70. Tasks schedule for the execution with one HS configuration memory.

Figure 71 presents another schedule for the same task-graph, but this time, with a LE memory instead of the HS one. We assume that the reconfiguration latency when using the LE memory is 50% larger than the reconfiguration latency when using the HS memory. In this second schedule, an overall execution delay has appeared because of the increment in the configuration latency.

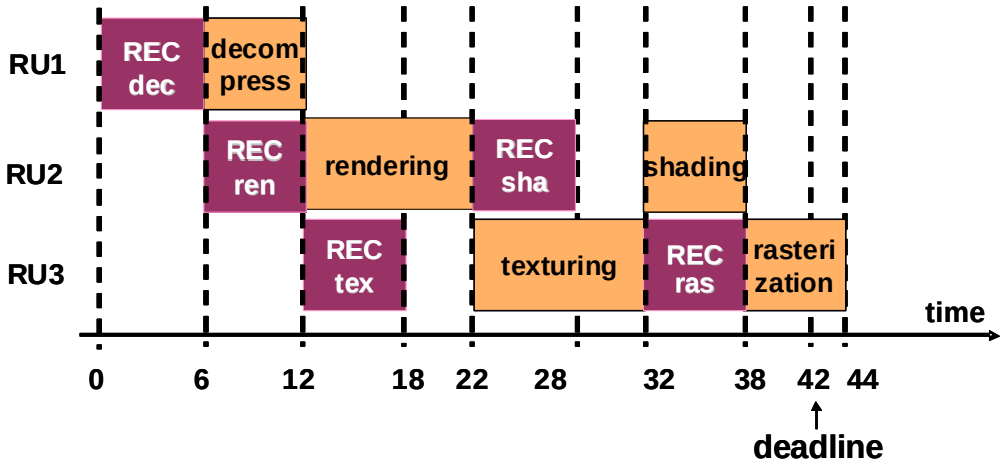


Figure 71. Tasks schedule for the execution with one LE configuration memory.

Finally, Figure 72 depicts the schedule obtained with the configuration memory hierarchy that we have proposed with an HS memory and an LE memory.

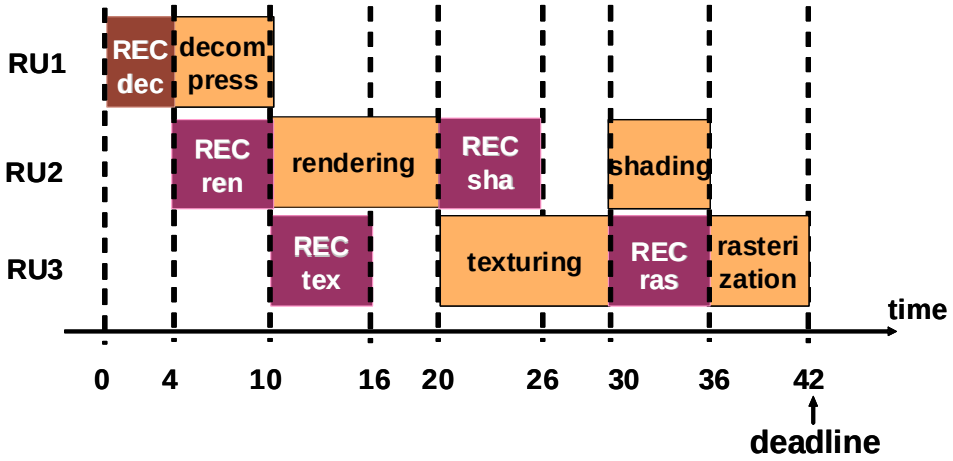


Figure 72. Task schedule for the execution with two configuration memories one High-Speed and other Low-Energy.

These memories have the same features as the HS and LE ones used in the previous execution examples. Our approach tries to achieve energy savings, moving tasks from HS memory to LE one, without reducing the overall system performance, while meeting all real-time constraints. From the resulting schedule, depicted in Figure 72, it is shown that our aim has been achieved: energy consumption has been

clearly reduced as three configurations have been mapped to the LE memory while obtaining the same performance as a system that only uses HS memories.

As it was explained in the previous chapter, we have developed a hybrid design-time/run-time reconfiguration manager designed to reduce the delays generated by the reconfigurations. This manager drives the reconfigurations of a set of RUs. The reconfiguration manager has been developed for a very simple configuration memory hierarchy, similar to the one depicted in Figure 64. In order to adapt our manager to a system with a memory hierarchy like the one proposed in Figure 68, a mapping algorithm must be included in the system. This module must decide whether configurations should be stored in the LE or in the HS memory. Storing a configuration in the LE memory reduces the energy reconfiguration overhead, but at the cost of a possible increase in the execution time.

The goal of our mapping algorithm is to identify a partition of the set of tasks that minimises the reconfiguration energy overhead without increasing the execution time overhead significantly. To achieve this goal, we have developed a systematic mapping algorithm that analyses the features of the task-graphs at design time and interacts with the prefetch module. This mapping algorithm is run once for each scenario generated for all the task-graphs that are going to be executed concurrently on the same reconfigurable platform. Then at run-time, when the system knows which tasks needs to load, these tasks will be loaded from the memory selected at design-time by the mapping algorithm.

6.4. Configuration mapping algorithm

An efficient prefetch technique may succeed hiding most of the task reconfigurations. In [RMVC05], our heuristic was able to hide at least 75% of them assuming that no reuse is present, which is the worst case possible. However, for certain tasks, it may fail meeting its objective because not always enough time is available to schedule all the loads in advance (e.g. task decompress in Figure 70).

In order to conveniently map the configurations to the different memories, it is very important to identify which tasks may generate delays due to their reconfiguration. The basic idea or our mapping algorithm is to assign those tasks to the HS memory and the remaining tasks to the LE memory. Our mapping algorithm uses as input the task schedule selected by the design-time scheduler for each of the task-graphs of the application. The pseudo-code of the algorithm that computes the design-time mapping is depicted in Figure 73.

```
for each task-graph do
{
        assign_2_HS;
        schedule_reconfigurations (&reference_schedule);
        assign_2_LE;
        schedule_reconfigurations (&current_schedule);
        compare (reference_schedule, current_schedule, &penalty, &task );

        while (penalty <> 0)
        {
                assign_2_HS (task);
                schedule_reconfigurations (&current_schedule);
                compare (reference_schedule, current_schedule, &penalty,
                        &task );
        {
}
```

Figure 73. Mapping algorithm pseudo-code.

Our mapping algorithm takes into account the weights calculated by the design-time scheduler for each task of the task-graph, that represents how critical the execution of each task is. As we have explained in the previous chapter, they are assigned by computing the longest path (in terms of execution time) from the beginning of the execution of the task to the end of the execution of the whole task-graph with an as-late-as-possible (ALAP) schedule. Hence, the first task in the critical path has always more weight than the others.

For each task-graph, the process starts by assigning all the configurations to the HS memory (*assign_2_HS*). That is the optimal mapping for performance. Next, it invokes the function *schedule_reconfigurations*, which applies a prefetch scheduling heuristic to obtain a schedule of the reconfigurations, assuming that none of the tasks assigned to RUs can be reused. Hence, all of them must be loaded. Any prefetch scheduling heuristic can be used. In our case, we use a branch and bound scheduling approach for small task-graphs and the heuristic presented in the previous chapter for large graphs. The obtained schedule is going to be used as a reference during the process.

Afterwards, the algorithm will look for a mapping with the same performance as the reference one, but with the maximum number of configurations assigned to the LE memory.

Our algorithm starts searching for an optimal mapping carrying out another schedule for the same task-graph, but this time assuming that all the configurations are stored in the LE memory (*assign_2_LE* + *schedule_reconfigurations*). Then, it compares both schedules identifying which tasks generate extra delays in the global

execution time when their configuration is stored in the LE memory instead of HS memory. The configuration selected to be mapped to the HS memory is, logically, the one with the greatest weight. This process is performed by the function *compare,* where *reference_schedule* and *current_schedule* are the two schedules to compare; and *penalty* is the difference in the execution time between these two schedules and *task* is the configuration selected.

After moving the first configuration from LE to HS, another schedule is computed assuming that all the configurations, except for the one previously selected, are assigned to the LE memory. This schedule is again compared with the reference one, and if extra delays are found, another configuration is assigned to the HS memory. This process continues iteratively until the execution time of the current schedule is the same as that of the reference schedule. At this moment, the algorithm generates the final partition, which provides as good performance as a partition that maps all the configurations to HS and also achieves clear energy savings as some of the configurations have been mapped to the LE memory. This algorithm can be easily modified to trade-off some delays for energy savings by changing the while condition from ‘while (*penalty* <> 0)’ to ‘while (*penalty* < *maximum_penalty_allowed*). However, in this work, we will assume the more demanding scenario in which no extra delay is accepted.

The next figures depict how the mapping algorithm works using a step by step example (from Figure 74 to Figure 78).

As we have mentioned before, the input of our mapping algorithm is the task-graph schedule selected at design-time. In this example, the scheduler has selected,

for those tasks that will be executed in the HW, in which unit of the reconfigurable HW they will be loaded.

In the following example the input task-graph has five tasks that have to be loaded and executed in a device with two RUs. We will assume that the configurations of these tasks can be preloaded in the on-chip configuration memories, but first we need to identify which tasks will be stored in the HS memory and which ones will be stored in the LE. This decision is also taken at design-time after applying our mapping algorithm. Figure 74 presents the selected schedule for our five tasks task-graph and the target architecture for this example.

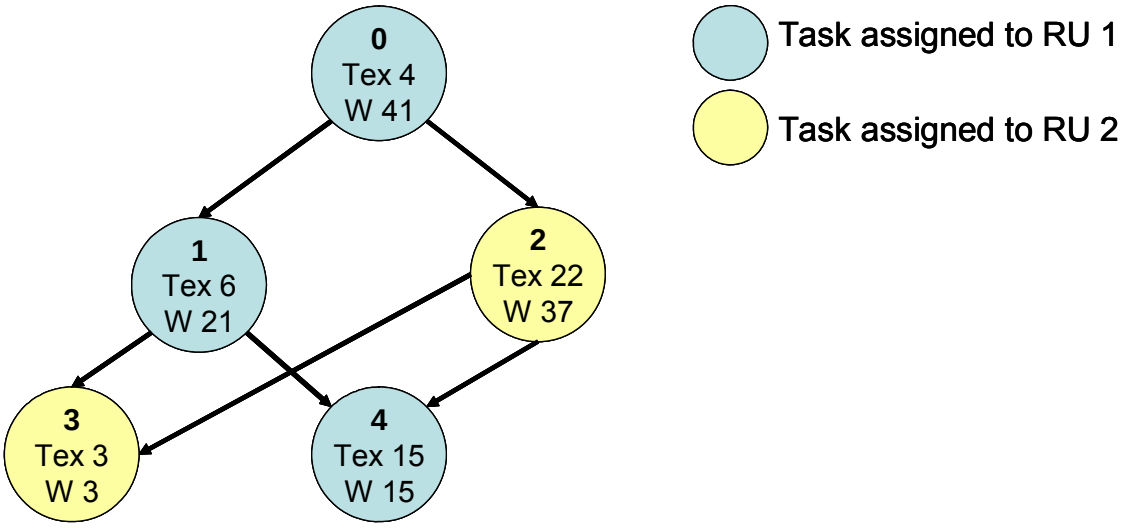


Figure 74. The selected schedule for the step by step mapping example task-graph.

In this example we realistically assume, taking into account the memory modules available in the market, that the memory module optimised for performance is 50% faster, but consumes 30% more energy per access than the one optimised to save energy. The time needed to load a task from an HS memory to an RU is for this example 4 time units (hence the loading latency for the LE memory is 6 time units).

During this example, as during the whole chapter, we take into account only the time and energy consumption due to the task movement between the on-chip memory modules and the RUs, assuming all the tasks to be executed in the reconfigurable HW are preloaded in the on-chip memory layer. In the next chapter we extend our approach to tackle the task load management between different levels of the memory hierarchy.

As we have mentioned before, the goal of our mapping algorithm is to identify a partition of the tasks that minimises the reconfiguration energy overhead without introducing any performance degradation. To accomplish this, the mapping algorithm identifies those tasks which load latencies cannot be hidden and maps them to the HS memory in order to minimise the reconfiguration overhead. The remaining tasks are mapped to the LE memory. Our mapping algorithm applies a prefetch schedule heuristic to obtain a schedule of the reconfigurations, assuming that none of the tasks assigned to RUs can be reused. Hence, all tasks must be loaded. Afterwards, at run-time, once the run-time scheduler have applied the reuse and replacement heuristic, only those tasks that must be loaded will be loaded from the memory selected by our mapping algorithm.

Before starting the analysis of this step by step mapping example, it is important to remember that current reconfigurable systems have only one reconfigurable circuitry to carry out the reconfigurations of the different RUs. Therefore, simultaneous reconfigurations are not supported.

The mapping algorithm starts assigning all the tasks for being loaded from the HS memory (*assign_2_HS*). Them it invokes the function *schedule_reconfigurations*,

which applies a prefetch heuristic, and obtains a schedule of the reconfigurations for the given task-graph. This is the optimal schedule for performance since all the tasks are loaded from the HS memory. This schedule is the reference schedule (*reference_schedule*) during the whole mapping process. Figure 75 depicts the reference schedule for the task-graph at figure 74 and its execution profile in terms of execution time and energy consumption.

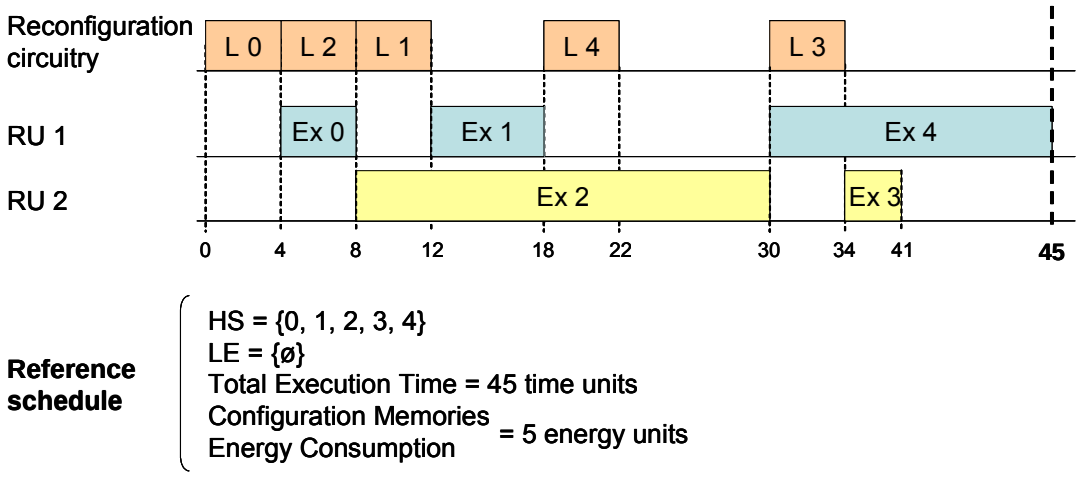


Figure 75. Reference schedule for the step by step mapping example (assuming all the tasks are stored in HS memory).

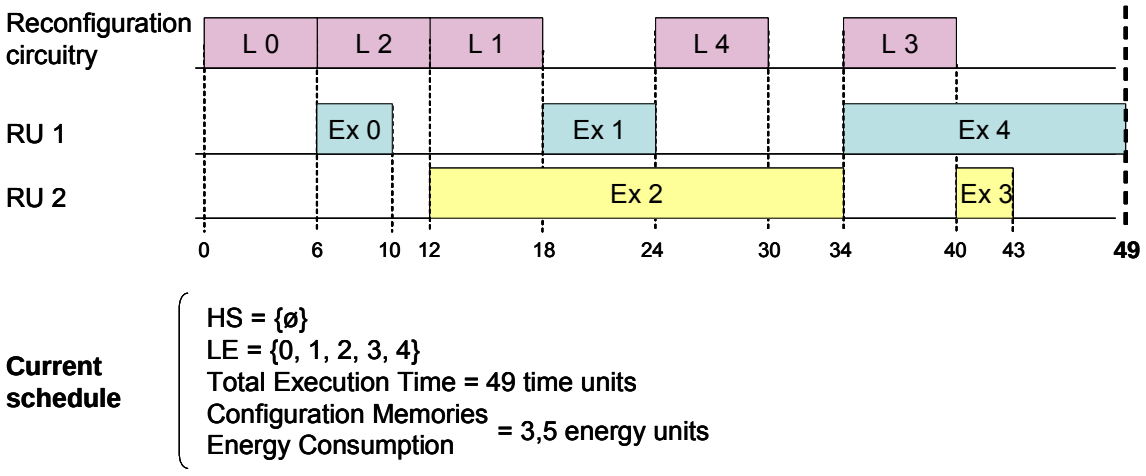


Figure 76. Initial current schedule for the step by step mapping example (assuming all the tasks are stored in LE memory).

After generating the reference schedule, the mapping algorithm computes the first current schedule (*current_schedule*) invoking again the function *schedule_reconfigurations*, but this time assuming that all the configurations are stored in the LE memory before (assi*gn_2_LE*). This reference schedule and its execution profile in terms of execution time and energy consumption are depicted in Figure 76.

In the next step the mapping algorithm compares the *reference_schedule* and the *current_schedule* (*compare*) and identifies the difference in the execution time between these two schedules. At this stage, this difference is 4 time units (*penalty*). The *Compare* function also identifies which tasks have generated this delay and selects one of them. The selected task is always the one of this set that has the greatest weight. In the following iterations this task will be assigned to the HS memory in order to reduce the delay (this is done in function *assign_2_HS(task))*. In this example Task 0 is selected.

Once the mapping algorithm identifies the task that has to be moved from the LE partition to the HS one, it invokes again the function *schedule_reconfigurations* to generate a new *current_schelude*. Figure 77 presents the resulting *current_schedule* and its execution profile once the task 0 load has been moved to the HS memory module.

As in the previous iteration, the mapping algorithm will compare (*compare*) this schedule with the *reference_schedule* and identify the differences. In this step the difference is 2 time units (*penalty*). As the time overhead of the *current_schedule* is

greater than zero, the mapping algorithm will again move a task from HS to LE (assi*gn_2_HS*). In this case the task selected is Task 2.

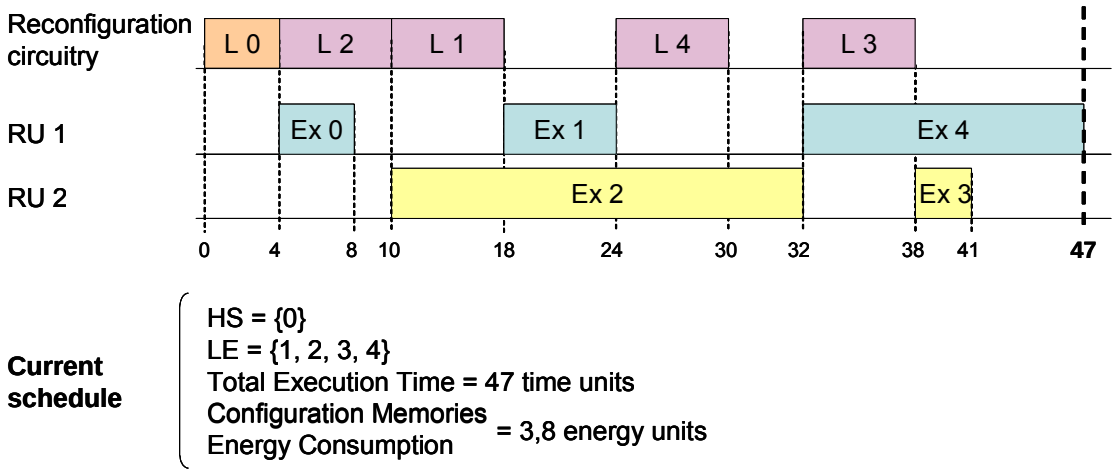


Figure 77. Iteration 1 current schedule for the step by step mapping example (once task 0 has been moved from the LE to the HS memory).

After this assignment, the algorithm invokes again the *schedule_reconfigurations* function to generate the new *current_schedule.* Figure 78 presents the resulting *current_schedule* and its execution profile.

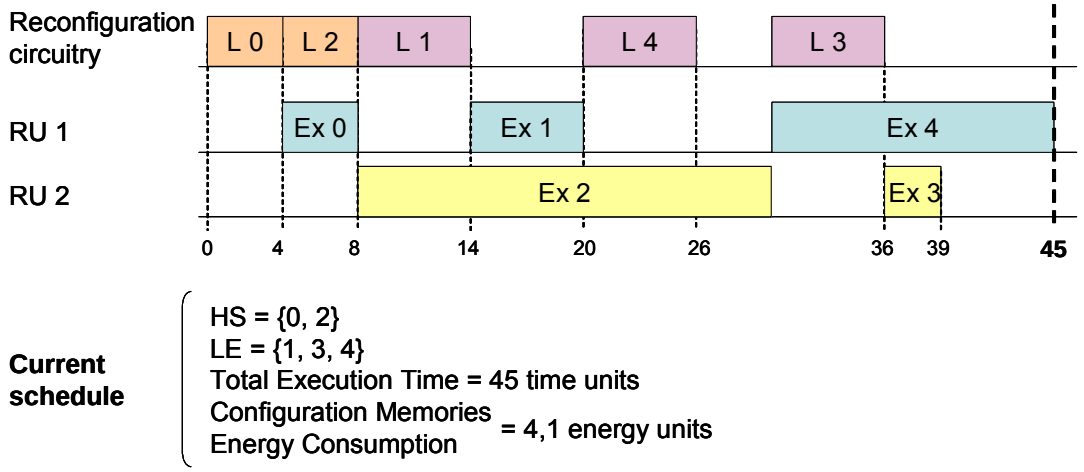


Figure 78. Iteration 2 current schedule for the step by step mapping example (once task 2 has been moved from the LE to the HS memory).

This time the comparison between the new *current_schedule* and the *reference_schedule* shows that both of them provide the same execution time. Hence this is the final mapping. In this case this partition achieves the same performance as the full HS system, whereas 66% of the tasks are stored in LE memories.

6.5. Experimental Results

To demonstrate our modules, we have integrated them into the scheduling environment presented in the previous chapters.

We have carried out two different representative experiments: one for a fine-grain reconfigurable platform and another for a coarse-grain platform.

In the first experiment we have evaluated our mapping technique with a set of multimedia applications assuming that all the tasks are executed in reconfigurable units implemented in a Virtex2 FPGA [XVirII], and that the task configurations can be stored either in a HS memory or in a LE memory. To carry out the experiment we have used the simulation environment presented in the previous chapters. The applications are a sequential and a parallel version of the JPEG decoder, an MPEG-1 encoder, and a Pattern Recognition application that applies the Hough transform over a matrix of pixels in order to identify geometrical figures.

To model the HS and the LE memories, we have used real data of two memory modules from ST microelectronics. In this realistic case, the memory module optimised for performance is 50% faster, but consumes 30% more energy per access

than the one optimised to save energy. In this experiment, we assume that the time needed to load a task from a HS memory to an RU is 4 ms (hence the loading latency for the LE memory is 6 ms) and that all the tasks are executed in the RU resources. These loading latencies are realistic for fine-grain reconfigurable resources.

In Table 6, the features of this set of applications and the results of our experiments are presented. **Ideal ex. time** is the average execution time of each application without reconfiguration overhead. **HS** and **LE time overhead** is the execution-time reconfiguration overhead when all the reconfigurations are mapped to HS and LE, respectively. Our reconfiguration manager has already been applied to reduce this overhead as much as possible. As it can be seen in Table 6, the increase in the reconfiguration latency has a direct impact on the execution time. Thus, if all the reconfigurations are stored in an LE memory instead of an HS memory, the execution time of the applications increases 9% on average. However, the energy consumed in order to read them is reduced by 30%. For many applications with demanding timing requirements, this increase in the execution time may not be acceptable. Hence, our mapping approach will try to achieve energy consumption reductions in the configuration memory hierarchy without introducing any execution-time penalisation.

The last four columns present the results of our approach. **Tasks** is the number of tasks of each task-graph. The following two columns depict how many of these tasks have been mapped to the HS memory and how many to the LE memory when applying our mapping algorithm. It is important to remark that this mapping guarantees the optimal performance (similar to the performance achieved when all

the configurations are stored in the HS memory). However, in the end around 65% of the configurations have been mapped to the LE memory. Hence, our mapping algorithm has achieved important energy savings in the configuration memory hierarchy without degrading the performance of the applications. It consumes on average 19.2% less energy per configuration, which is close to the theoretical maximal saving of 30% with the given memory library. If the LE version of the memory is further optimised, we can expect even larger savings. A good example of such ultra-low energy SRAMs is provided in [CoSC08].

Application	***Ideal ex. time***	***HS time overhead***	***LE time overhead***	***Tasks***	***Our mapping***		***Energy Rec. overhead***
					HS	***LE***	
Pattern Rec.	94 ms	+4%	+6%	6	1	5	-25%
JPEG dec.	79 ms	+5%	+8%	4	1	3	-22.5%
Parallel JPEG	54 ms	+7%	+26%	8	5	3	-11.3%
MPEG enc.	37 ms	+16%	+27%	5	2	3	-18%
Average	**66 ms**	**+8%**	**+17%**				**-19.2%**

Table 6. Features of the set of multimedia benchmarks and experimental results for fine grain reconfigurable HW.

Application	***Ideal ex. time***	***HS time overhead***	***LE time overhead***	***Tasks***	***Our mapping***		***Energy Rec. overhead***
					HS	***LE***	
DSP_dot_prod	32 µs	+19%	+28%	3	1	2	-18%
DSP_vec_sumq	32 µs	+19%	+28%	2	1	1	-18%
DSP_q15_tofl	5645 µs	+0%	+0%	3	1	2	-10%
DSP_neg32	109 µs	+6%	+8%	3	1	2	-21%
DSP_min_val	114 µs	+5%	+8%	3	1	2	-20%
DSP_dotp_sqr	32 µs	+19%	+28%	3	1	2	-18%
DSP_blkmove	9 µs	+125%	+191%	2	2	0	-18%
Average	**32 µs**	**+27%**	**+42%**				**-18%**

Table 7. Features of the set of DSP benchmarks and experimental results for coarse-grain reconfigurable HW.

In order to evaluate our approach for a coarse-grain platform we have used the coarse-grain simulation environment CRISP presented in chapter 2. This platform is also very similar to the domain-specific VLIW architectures that become very popular for low-energy high-performance wireless and multi-media application kernels. In this environment the designer can define the coarse-grain architecture and the memory hierarchy. In our case, we have defined an architecture with four coarse-grain reconfigurable units. Each unit has a set of programmable ALUs, and an internal memory of 3KB that stores its current configuration. As in the previous experiment, configurations are loaded either from a HS memory or from a LE memory. The loading latency is 6 μs if the configuration is stored in the HS speed memory and 9 μs if it is stored in the LE memory. Hence, in this experiment the configuration latency is drastically smaller than in the previous one. Therefore, we will apply our mapping technique for a smaller task granularity. In this case, we have selected a set of DSP benchmarks developed by Texas Instruments [TeIn09]. The details of these experiments are depicted in Table 7.

For this task granularity the impact of the reconfiguration latency is even greater than in the previous experiment. As a result, the mapping algorithm has assigned a smaller percentage of tasks (58%) to the LE memory. However, it is also a good result, since our memory hierarchy still provides the same performance as a fully HS approach, while significantly reducing the energy reconfiguration overhead (by 18% in this experiment).

To sum up, reconfigurable HW offers high performance and flexibility at the cost of a reconfiguration overhead both in execution time and energy consumption.

Previous work [RVMV04] has demonstrated that, with the appropriate support, the execution-time overhead can be drastically reduced. However, since embedded systems are frequently battery-dependent, specific support is needed to reduce the reconfiguration energy overhead. In this chapter we propose a simple memory hierarchy extension for configurations with a memory module optimised for performance and another module optimised to reduce the energy consumption per access. In addition, we have developed a systematic mapping algorithm to decide where to store each configuration and we have integrated it in our previous reconfiguration manager. Our mapping algorithm attempts to store the maximum number of configurations in the memory optimised for energy without generating any performance degradation.

Clearly, the benefits of our approach depend on the granularity of the tasks and on the configuration latency. If for some given values the reconfiguration overhead is not significant, our mapping technique will not be needed. However, as it can be seen in our experiments, in many cases the reconfiguration overhead has a significant impact both on energy consumption and performance. In these situations our approach achieves relevant energy savings without decreasing the performance. Thus, in our experiments, our algorithm found an optimal mapping for performance, while consuming about 20% less energy per configuration loaded.

Our experiments have been carried out for both fine-grain and coarse-grain reconfigurable resources with a centralised memory hierarchy assuming that the needed configurations have been previously stored in one of the on-chip configuration memory following our mapping algorithm. However, if the number and

size of the configurations exceeds the size of the configuration memory this assumption will not be correct. In the next section, we extend our approach to support systems where not all the configurations can be pre-stored in the configuration memory.

Chapter 7:
Configuration Memory Hierarchy Extension

In chapter 6 we introduced the idea of including a configuration memory hierarchy with an on-chip dual memory layer (HS/LE). Initially, we assumed that before starting the execution of a new task-graph, all the corresponding configurations have been preloaded in the on-chip configuration memory. This assumption should be acceptable for most of our target applications because only the most computing intensive kernels are typically assigned to the reconfigurable units (RUs). Nevertheless, depending on the applications and the target architectures, this assumption may not be correct. In this chapter we present an extension of the approach presented in chapter 6 to tackle the new challenges that appear in the HS/LE configuration memory hierarchy system management when all the tasks assigned to HW cannot be preloaded into the on-chip memory layer due to size

limitations. The most important reasons to restrict that on-chip size are reduced area and energy consumption.

In the previous chapter, we were only considering static systems, in which only one task-graph was executed several times. In this chapter, besides taking into account the capacity of the on-chip memory modules, we will target, not only static, but also dynamic systems where several task-graphs can be executed interleaved. In fact we will consider two different problems, and we will adapt our previous work to both of them.

In the first problem only one task-graph is active in the platform, and the objective is to optimise the mapping of the different configurations of the graph in a similar way as in the previous chapter, but taking into account that sometimes not enough space will be available in the on-chip level. In those cases, some configurations will be always loaded from the external memory in order to prevent a configuration trashing problem (they will be tagged for the mapping algorithm as “non-cacheable” tasks). The objective of this simple extension is to identify which are the optimal candidates to be left in the external memory in order to optimise the performance, and minimise the energy consumption.

In the second case, we will target dynamic systems were several task-graphs may be event-driven (semi-randomly potentially) executed at run-time. We assume that these task-graphs will be executed several times, but we do not how many graphs and which ones will be active concurrently. Since we do not know the actual running conditions, we cannot look for an optimal solution at design-time. Hence, in this case we will look for the solution that meets a given performance constraint, as

long as no run-time conflicts are present, while generating the minimum pressure on the on-chip configuration memory. In addition, we will attempt to reduce the energy consumption, mapping conveniently those tasks assigned to the on-chip level. In this case, we do not want only to optimise the execution of a single graph, but also attempt to reduce possible conflicts in the on-chip level, since these conflicts may generate both important delays in the execution and energy penalisations.

In order to evaluate our approach, the first problem that we faced was to estimate the energy consumption and the latency of the external memories. When we analysed different commercial boards, we found different memory technologies that can be used to store the configurations: flash memories, EEPROMs, DDR2 RAMs... The actual numbers for the external memories will drastically vary from one board to another. In addition, we could not always find the information we need for the estimations, since energy numbers are normally not available in the documentation of the board. Hence, we decided to use again CACTI to obtain some significant numbers, since recently CACTI has included support to estimate the energy consumption and access latency of DRAMs [CACT08].

The Figure 79 depicts an example of the energy/latency trade-off obtained using CACTI. It includes the data regarding the latency and the dynamic energy consumption per read port for DRAM memory modules of different sizes (1 Mbytes, 2 Mbytes, 4 Mbytes, 8 Mbytes, 16 Mbytes, and 32 Mbytes). The results have been normalized using the data obtained by CACTI for a 64 Kbytes HP SRAM memory module. We have obtained these numbers using the same parameters presented in Table 5.

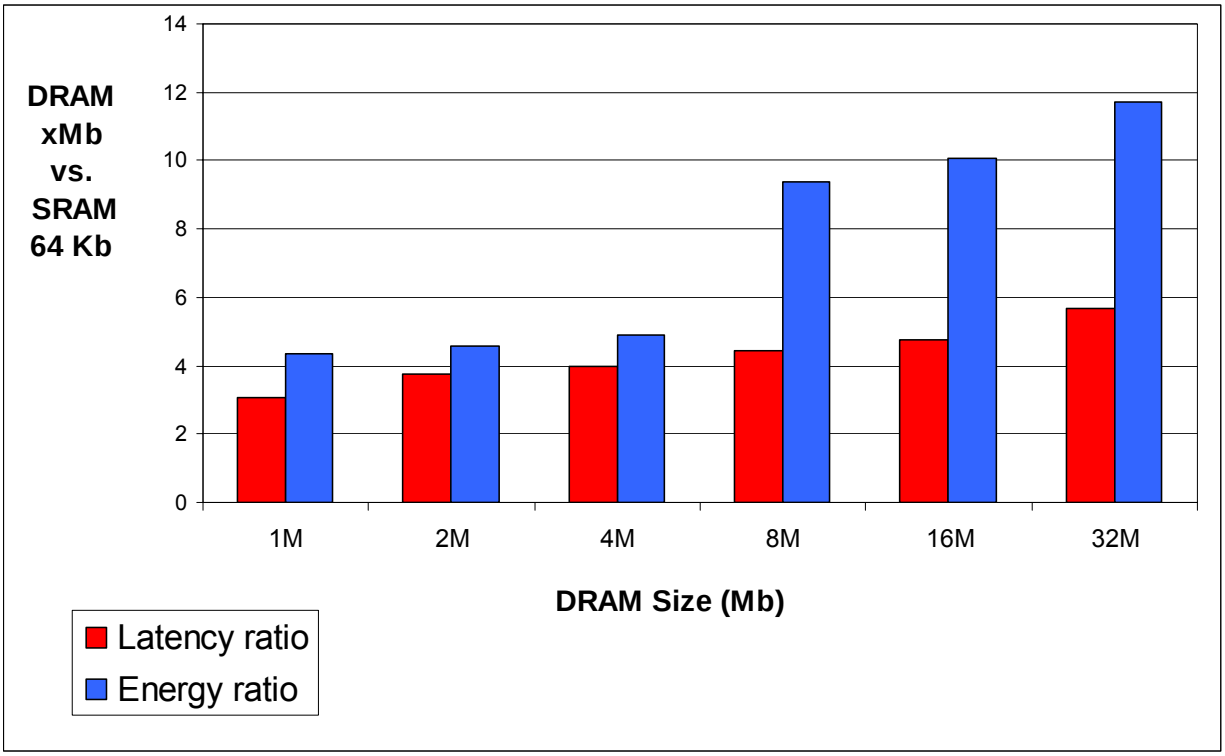


Figure 79. Access latency and energy consumption per access for different DRAM memories obtained with CACTI. The results have been normalized using the data of a 64 Kbytes HP SRAM memory.

As it can be seen in Figure 79 the latency of a DRAM memory of 1 Mbytes is three times worse than the latency of a HP-SRAM memory of 64 Kbytes, and the energy consumption is four times worse. For memories of 2 or 4 MB we have obtained trade-offs that are very similar, and for memories with greater capacities, the external memory is more or less five times slower and consumes ten times more energy. We believe that any of these memories could be used to test our approach. We have selected the first one, since it represents the case where the external memory is more efficient. We will compare our approach with a system that always carries out the reconfigurations fetching the new configuration from the external memory. Hence, if our approach introduces important benefits when compared with

the most efficient external memory, it will clearly introduce benefits if we consider any other scenario that uses less efficient memories.

7.1. Motivational Example

In this section, we use a step by step example to explain in detail the challenges that our approach faces when all the tasks assigned to the reconfigurable elements cannot be preloaded into the on-chip memory layer. In fact we will go back to the example described in chapter 6 but assigning a maximum capacity to the on-chip memory modules.

Figure 80 depicts the selected schedule of the task-graph for the target architecture for this example. The application represented by the task-graph has five tasks that have to be loaded and executed in a device with two RUs.

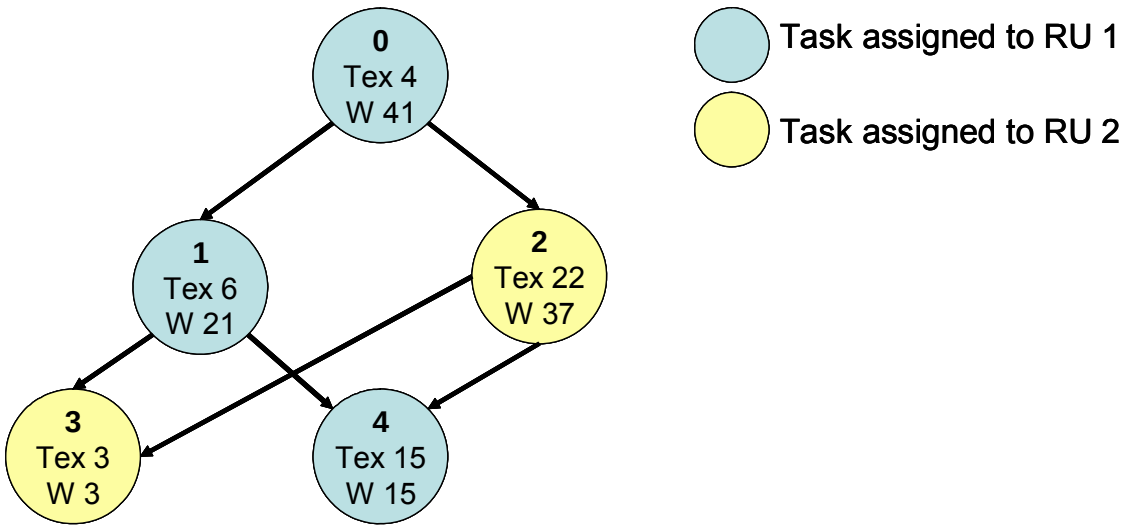


Figure 80. The selected schedule for the step by step mapping example task-graph.

As we have mentioned before, in chapter 6 we assumed that the configurations of all tasks can be preloaded in the on-chip configuration memories. Now, we will illustrate what happens when this is not possible. In this example, we assume that only four tasks can be stored in the on-chip HS/LE configuration memory layer: two tasks in the HS module and two tasks in the LE module.

Taking into account the results obtained using the chapter 6 mapping algorithm for this graph, the optimal task mapping consists of allocating Task 0 and Task 2 into the HS memory and Task 1, Task 3 and Task 4 into the LE memory module. As we have mentioned before, now, only two tasks can be stored in each on-chip memory module. Thus, one of the tasks assigned to the LE memory module cannot be preloaded. In this case Task 3 is the one that is not preloaded, since it is the less critical task (with the lowest weight W) of those assigned to the LE. Thus, before starting the execution of the task-graph, in an initialization phase, Task 0 and Task 2 are preloaded into the HS memory module, and Task 1 and Task 4 into the LE memory. When Task 3 is requested, it will produce an on-chip configuration memory miss. Hence, Task 3 will be loaded from the external memory into the assigned RU, and at the same time it will be stored into the on-chip memory layer, in the LE memory module, as well.

We will now analyse the execution of the task-graph in detail. Taking into account the state of the on-chip memory layer, in the first iteration of the task-graph, Task 0, Task 1, Task 2 and Task 4 are loaded into the RUs from the on-chip memory layer. Figure 81 depicts the first iteration execution profile and the state of the on-chip configuration memory modules until the end of the execution of Task 2. As indicated

in chapter 6, we realistically assume, taking into account the memory modules available in the market, that the typical memory module optimised for performance is 50% faster, but consumes 30% more energy per access than the one optimised to save energy. The time needed to load a task from an HS memory to an RU is for this example 4 time units (hence the loading latency for the LE memory is 6 time units).

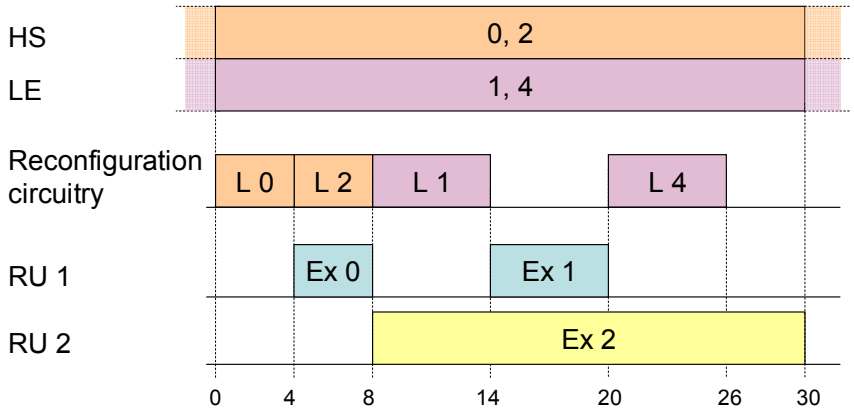


Figure 81. Task-graph first execution iteration, until Task 2 execution ends.

At this point, when Task 2 execution ends, Task 3 has to be loaded from the external memory. Now, we have to decide whether to load it also in the configuration memory applying a replacement policy, or to use the on-chip configuration memory only for the preloaded initial tasks. If we are dealing with dynamic systems the second option may not be efficient. Hence, in this example we will assume that when a new task is loaded in the system, its configuration is also fetched into the configuration memory. Hence the configuration memory will operate in a similar way as a two-set associative cache, with one set for the task assigned to HS and the other for the tasks assigned to LE. We have chosen the well-known LRU (Least Recently Used) replacement policy to select the victim configuration that is replaced by the new one.

In this policy the victim configuration is the task that has been used further away in the past. In this example the victim configuration is Task 1.

Figure 82 depicts the execution profile of the first iteration of the task-graph, and the state of the on-chip configuration memory modules. In this example, we suppose that when a task is loaded into a RU it is also fetched into the on-chip configuration memory layer in parallel. In order to model the external memory, we have assumed, taking into account the memory modules available in the market, that the external memory is three times slower that the memory module selected to implement the HS on-chip memory (hence the loading latency for the external memory is 12 time units).

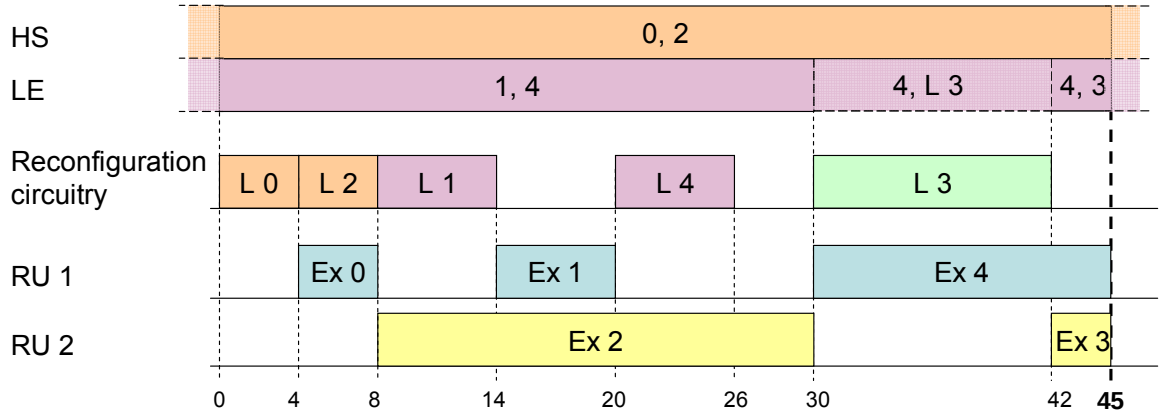


Figure 82. Task-graph first iteration.

As it can be seen in Figure 82, after the first iteration of the task-graph, in the on-chip memory layer the HS memory module stores the configurations of Task 0 and Task 2, and the LE memory module stores the configurations of Task 4 and Task 3. Hence, at the beginning of the next iteration of the task-graph, Task 1 has to be fetched from the external memory. When Task 1 is loaded into the RU it is also

fetched into the LE memory module of the configuration memory. Since the LE memory module can only store two tasks at the same time, the LRU policy is applied again. This time the victim task is Task 4.

However, Task 4 will be needed soon, generating a new miss in the configuration memory. Thus, it has to be loaded into the RU from the external memory, and it is also fetched into the LE memory module. Since the LE memory module is full, the LRU policy is applied again. This time the victim task is Task 3. Before finishing the second iteration Task 3 is requested. Hence a new miss must be handled. And in this case the victim configuration is Task 1.

Figure 83 depicts the execution profile of the second iteration of the task-graph, and the situation of the on-chip configuration memory modules during it. The execution time for this iteration is 53 time units, which is 8 time units worse than the first iteration. Moreover, three replacements have been done, with the corresponding energy penalisation.

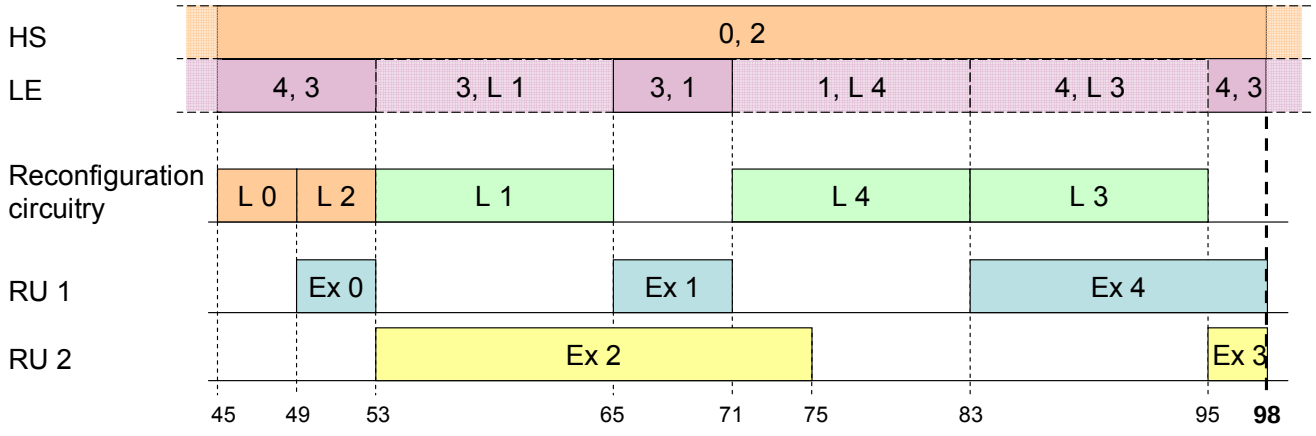


Figure 83. Task-graph second iteration.

As it can be seen in Figure 83, after the second iteration of the task-graph, in the on-chip memory layer the HS memory module stores Task 0, and Task 2 and the LE memory module stores Task 4, and Task 3. Hence, the on-chip configuration memory layer has the same state as at the end of the first iteration. Therefore, the subsequent iterations will have the same execution profile as the second iteration.

Hence, if this task-graph is executed several times, all the subsequent iterations will have the same execution profile. So Task 1 is always replaced by Task 4, which is always replaced by Task 3, which will be replaced by Task 1. This effect of configuration being fetched and replaced systematically on a cache memory is called thrashing, and it is well known in the memory management community.

The configuration thrashing effect that appears in the on-chip configuration layer is due to the high pressure that the configuration places on this part of the configuration memory hierarchy. In this example the pressure is especially high on the LE module. Several options exist trying to relieve the pressure from the on-chip configuration layer. One simple solution is to enlarge it adding space to store more configurations. Nevertheless, this is not always possible, since the size of these modules is limited by the size of the chip and moreover to enlarge the memory modules also increases the energy consumed per access.

Another possible solution is to establish restrictions at the task-graph level in order to guarantee that the configurations assigned by each graph to the on-chip modules do not exceed their capacity. This is the first problem we will target. To this end we will describe a relatively straightforward extension of our previous mapping algorithm that attempts to accomplish the same objective as the chapter 6 mapping

algorithm: obtain a task partition that provides the best possible performance (the objective is to provide as good performance as will do a system that only includes HS modules), while reducing as much as possible the energy consumption. The main difference in this case, is that the algorithm takes into account that the size of the configurations allocated into each module cannot be greater than the size of these modules. With this approach at the beginning of the execution of a task-graph some selected configuration will be preloaded to the on-chip configuration module, and the remaining ones will be always fetched from the external memory without loading them in the on-chip configuration memory. Therefore, the tasks selected by the mapping algorithm for being preloaded into the on-chip configuration memory at design-time will never be replaced during the task-graph execution.

As we have mentioned before, this extension will efficiently tackle the new mapping challenges only for systems with one active task-graph. However, if the system supports the interleaved execution of several graphs, the configuration trashing problem may still have an important impact in the system performance. For this problem, we propose another solution. Basically, each graph will be analysed separately at design-time (since at design-time, we do not know how the different task-graphs will be interleaved at run-time) to identify those configurations that can be loaded from an external memory without introducing a performance degradation in the whole task-graph execution, or those that will introduce a performance degradation that can be accepted by the system. Once these configurations are identified, they will be tagged as “non-cacheable”, i.e. they will be always loaded from the external memory, reducing the pressure on the on-chip configuration memory.

Therefore, the target of the mapping approach for dynamic systems is to obtain a task partitioning that meets a required performance level, while assigning the minimum number of tasks to the on-chip configuration memory layer. In addition, those tasks assigned to on-chip configuration memory will be partitioned between the HS and the LE modules in order to reduce the energy consumption.

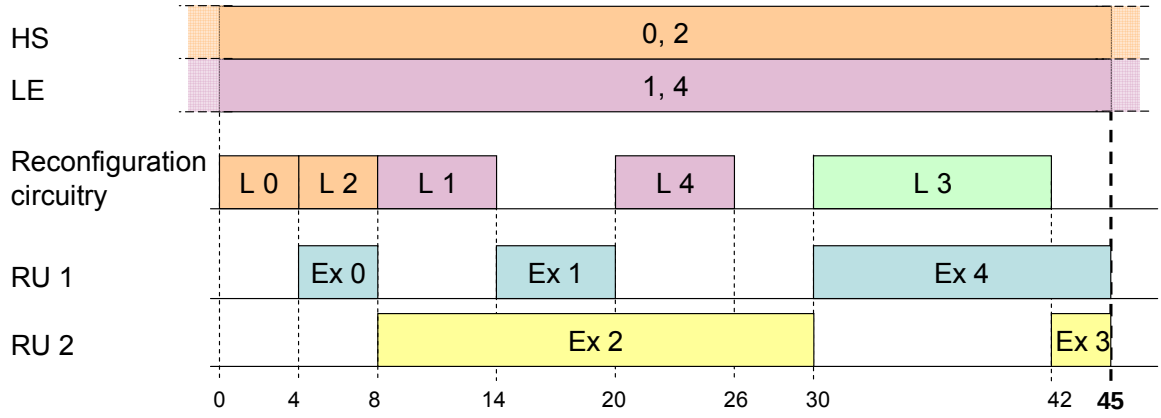


Figure 84. Task-graph iterations, loading Task 3 from external memory and assuming that Task 0, 1,2 and 4 were already loaded in the on-chip configuration memory.

Continuing with the motivational example, in this case only one graph is active. Hence, we will use the first solution. The mapping process will analyze the task-graph taking into account the selected schedule and the size of each one of the on-chip memory modules (each on-chip memory module only can store two configurations), identifying that Task 3 is the best candidate to be fetched form the external memory. Hence, it will assign Task 0 and Task 2 to the HS module, Task 1 and Task 4 to the LE, and finally Task 3 to the external memory. In addition Task 3 is tagged as “non-cacheable”; hence it will not be fetched to the configuration memory when dealing with a miss. Figure 84, presents the results of this approach. In this case, loading

Task 3 from external memory avoids the configuration thrashing problem, and the execution time of whole task-graph is the optimal one, i.e. the same execution time that would be obtained if all the configurations were stored on a HS memory.

7.2. Configuration mapping algorithm extension for static systems

As we have mentioned before, in this chapter we extend our previous approach in order to take into account the capacity of the on-chip memory modules. In this extension we consider that tasks can be loaded not only from the HS or the LE on-chip memory modules, but also from external memory. In this section, we proposed an extension of the chapter 6 mapping algorithm for systems in which only one active task-graph is executed on the platform. Hence, we assume that this graph is going to be executed several times without sharing any resource with any other task-graph.

This new mapping algorithm shares the objective of the previous one: obtain a task partition that provides the best possible performance, while reducing as much as possible the energy consumption. The main difference is that this algorithm takes into account the size of the on-chip modules. Since the on-chip memory is typically both better in performance and energy consumption than the external one, this algorithm will always attempt to maximise the use of the on-chip memory layer, assigning as many configurations as possible. If not enough space is available for all the configurations in the on-chip memory level, the algorithm will assign some of them to

the external memory in order to avoid a configuration trashing problem. The pseudo-code of this mapping algorithm is depicted in Figure 85.

Basically, the new mapping algorithm starts as the previous one, identifying which are the configurations that generate delays in the execution unless they are loaded from a HS memory, and assigning all of them to the HS on-chip memory module, whereas the remaining configurations are initially assigned to the LE module (*chapter6_mapping(&reference_schedule)*).

In the next step, the algorithm checks if enough space is present in the HS module for all the configurations assigned to it (*size*(*HS_tasks*) > *size*(*HS*)). If not enough space is available, those configurations with less weight are assigned to the LE module (*assign_2_LE*).

Afterwards, the algorithm checks if enough space is present in the LE module for all the configurations assigned to it (*size(LE_tasks)* > *size(LE)*). If not enough space is available, the algorithm will check if some space is available in the HS module, and in this case those tasks with more weight will be moved from LE to HS (*assign_2_HS*). If this is not enough, i.e. if the HS and the LE on-chip memory modules are full, the configurations with less weight are assigned to the external memory (*assign_2_Ext*) and they are tagged as “non-cacheable” tasks, in order to avoid a trashing problem in the on-chip modules.

To illustrate how the new mapping algorithm works, we are again going to apply it to the motivational example of this chapter. There the application represented by the task-graph has five tasks that have to be loaded and executed in a device with

two RUs. Figure 80 depicts the selected schedule of the task-graph for the target architecture for this example.

```
chapter6_mapping(&schedule);

if size(HS_tasks) > size(HS) then
{
        list_HS_tasks= sort_HS_tasks (schedule);
        first = first (list_HS_tasks);
        while size(HS_tasks) > size(HS)
        {
                assign_2_LE (list_HS_tasks[i]);
                first = move_right (first);
        }
}

if size(LE_tasks) > size(LE) then
{
        list_LE_tasks = sort_LE_tasks (schedule);
        first = first (list_LE_tasks);
        last = last (list_LE_tasks);
        while (size(HS_tasks) < size(HS)) and (size(LE_tasks) > size(LE))
        {
                assign_2_HS (first);
                first = move_right (first);
        }
        while size(LE_tasks) > size(LE)
        {
                assign_2_Ext (last);
                last = move_left (last);
        }
}

schedule_reconfigurations (&schedule);
```

Figure 85. Mapping algorithm for static system pseudo-code.

The new mapping algorithm starts considering the results obtained for the chapter 6 mapping algorithm. The output of this step assigns Task 0 and Task 2 to the HS memory and Task 1, Task 3 and Task 4 to the LE module. After generating the initial mapping, the new mapping algorithm checks if enough space is present in

the on-chip memory. In this example only two tasks can be stored in each on-chip memory module. Thus, enough space exists for the two tasks assigned to the HS memory module, but one of the tasks assigned to the LE memory module cannot be allocated there. Hence, the mapping algorithm has to reallocate it.

Firstly, the mapping algorithm looks for free space in the HS module since it attempts to maximise the use of the on-chip modules. However, as the HS on-chip memory module is already full, the configuration of Task 3 is assigned to the external memory. The algorithm selects this task because it has less weight. After this reallocation, the algorithm checks if still the configurations assigned to LE exceed its capacity, but since currently only two configurations are assigned to LE, no further movements are needed. Hence, it will assign Task 0 and Task 2 to the HS module, Task 1 and Task 4 to the LE, and finally Task 3 to the external memory. In addition Task 3 is tagged as “non-cacheable”. It means that Tasks 3 will not be brought to the on-chip configuration memory modules when dealing with a miss. This is exactly the case that was depicted in Figure 84. In this case, the algorithm has found an optimal mapping regarding performance, since the execution time is similar to the one obtained with a system that only uses a HS memory.

This mapping algorithm obtains optimal or near-optimal results in performance, and attempts to take full advantage of the on-chip memory modules to achieve important energy savings. However, it can only achieve these objectives when only one task-graph is active in the platform. When several task graphs are active, this solution may not be efficient, and the trashing problem may arise. For this situation we have developed another mapping algorithm.

7.3. Configuration mapping algorithm extension for dynamic systems

The problem of the previous algorithm is that the decision of allocating as many tasks as possible in the on-chip memory layer is optimal for one task, but can lead to a trashing problem when several task-graphs are active at the same time.

In this section we propose another mapping algorithm targeting dynamic systems that supports the interleaved execution of several graphs. We assume that these task-graphs will be executed several times, but we do not know how many active graphs will be active concurrently. Since in this scenario we do not know the actual running conditions, we cannot look for an optimal solution at design-time. Hence, in this case the mapping algorithm looks for the solution that meets a given performance constraint, as long as no run-time conflicts are present, while generating the minimum pressure on the on-chip configuration memory. In addition, it attempts to reduce the energy consumption, mapping conveniently those tasks assigned to the on-chip memory layer. In this case, the mapping algorithm does not only optimise the execution of a single task-graph, but also attempts to reduce possible conflicts in the on-chip memory layer, since these conflicts may generate both important delays in the execution and energy penalisations.

Basically, each graph will be analysed separately at design-time (since at design-time, we do not know how the different task-graphs will be interleaved at run-time) to identify those configurations that can be loaded from an external memory without introducing a performance degradation in the whole task-graph execution, or

those that will introduce a performance degradation that can be accepted by the system. Once these configurations are identified, they will be always loaded from the external memory, again being tagged as “non-cacheable” tasks. In this way, the mapping algorithm reduces the pressure on the on-chip configuration memory. Therefore, the target of this mapping approach is to obtain a task partitioning that meets a required performance level, while assigning the minimum number of tasks to the on-chip configuration memory layer. In addition, those tasks assigned to on-chip configuration memory layer will be partitioned between the HS and the LE modules in order to reduce the energy consumption.

As in the previous section, the new mapping algorithm is based on the chapter 6 mapping algorithm, and it basically consists in applying it to the different levels of the configuration memory hierarchy recursively. In this thesis, we are only considering two levels: HS/LE and external, but this algorithm can be easily generalized to N levels.

The pseudo-code of the new mapping algorithm that computes the design-time mapping for dynamic systems is depicted in Figure 86.

As the previous algorithms, this one needs as input the task schedule selected by the design-time scheduler for each task-graph, and the weights of each node. As we can see in figure 86, the new mapping process works in two steps.

```
for each task-graph do
{

        assign_all_tasks_2_HS;
        schedule_reconfigurations (&reference_schedule);
        assign_all_tasks_2_LE;
        schedule_reconfigurations (&current_schedule);
        compare (reference_schedule, current_schedule, &penalty, &task );

        while penalty <> 0 and (size(HS_tasks) + size(task)) > size(HS)
        {
                assign_2_HS (task);
                schedule_reconfigurations (&current_schedule);
                compare (reference_schedule, current_schedule, &penalty,
                &task );
        }
        # Step 1 Results. Cacheable tasks have been partitioning in two categories:
        #                       - category 1: Tasks preloaded in HS
        #                       - category 2: Tasks preloaded in LE

        # Step 2. It applies again the same algorithm to partition the tasks assigned in step 1 to LE
        #     in two categories: those that will remain in LE and those that will be assigned to the
        #     external memory.

         reference_schedule = current_schedule;
        assign_LE_tasks_2_ExtMem;
        schedule_reconfigurations (&current_schedule);
        compare (reference_schedule, current_schedule, &penalty, &task );

        while penalty <> 0 and (size(LE_tasks) + size(task)) > size(LE)
        {
                assign_2_LE (task);
                schedule_reconfigurations (&current_schedule);
                compare (reference_schedule, current_schedule, &penalty,
                &task );
        }

        # Step 2 Results. Tasks have been partitioning in three categories:
        #                       - category 1: Tasks preloaded in HS
        #                       - category 2: Tasks preloaded in LE
        #                       - category 3: Tasks to be loaded from External Memory
        #
}
```

Figure 86. Mapping algorithm pseudo-code.

The first step identifies those tasks that must be stored in the HS on-chip memory in order to meet the given execution time constraint. If no execution time constraint is given, the algorithm attempts to achieve the maximum performance (i.e.

the performance obtained when all the configurations are stored in HS memories). This step basically consists in applying the algorithm explained in the previous chapter but taking into account the size of the tasks to be loaded and the size of the on-chip memory modules. Hence, as we explained in chapter 6, the mapping process first step starts assigning all the configurations into the HS memory (*assign_all_tasks_2_HS*). This is the optimal mapping for performance, and it will be the performance objective to achieve for the mapping process unless a given execution time constraint is given. In our experiments we will always attempt to obtain the optimal performance since it is the more demanding scenario. Of course this will limit the potential reductions of the energy consumption, but even in this exigent scenario our mapping techniques will achieve significant energy-savings.

After generating the optimal mapping for performance, the first step will look for a mapping with the same performance as the reference one, but with the maximum number of configurations assigned to the LE memory. This is the same iterative process described in chapter 6. At the end of this process some tasks will be assigned to HS whereas the others will be assigned to LE. If the number of tasks assigned to HS exceeds its capacity, those with less weight will be moved to the LE module.

The second step of the mapping algorithm starts from the output of the first step. It attempts to identify those tasks of the category 2 that can be loaded from the external memory without introducing a performance degradation in the task-graph execution, or those that will introduce a performance degradation that can be accepted by the system. Basically, the second step consists in applying again the

iterative process described in chapter 6 but this time partitioning the tasks initially assigned to LE between LE and the external memory. The idea is to find a mapping with the same performance as the reference one, but with the maximum number of configurations assigned to the external memory.

It starts generating a new schedule assuming that all the configurations that at the reference scheduling are stored in the LE memory, are now fetched from the external memory (*assign_LE_tasks_2_ExtMem* + *schedule_reconfigurations*). Then, it compares both schedules, identifying which tasks generate delays. Then it selects the one with the greatest weight and assigns it to the LE module. After moving the first configuration another schedule is computed assuming that all the configurations, but the one previously selected, are assigned to the external memory. This schedule is again compared with the reference one, and if an extra delay exists, another configuration is assigned to the LE module. This process continues iteratively until the execution time of the current schedule is the same as that of the reference schedule, or until no more space is available in the LE module ((*size*(*LE_tasks*) + *size*(*task*) > *size(LE)*).

The next figures (from Figure 87 to Figure 90) depict how the new mapping algorithm works with the same example introduced at the beginning of this chapter. This task-graph has five tasks that have to be loaded and executed in a device with two RUs. As in the motivational example, we will assume that only four tasks can be stored in the on-chip HS/LE configuration memory layer: two tasks in the HS module and two tasks in the LE module.

As we have mentioned before, the first step of the new mapping algorithm consists in applying the algorithm presented in the previous chapter. The result is depicted in Figure 87.

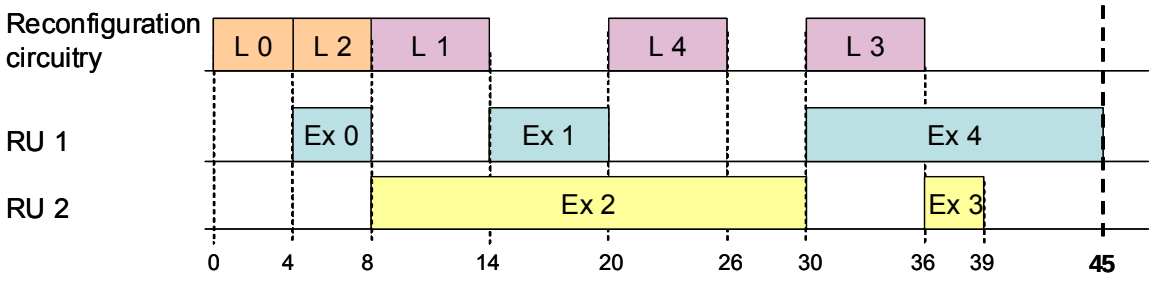


Figure 87. Reference schedule for the second step of the mapping.

The second step starts assuming that all the tasks initially assigned to LE are now assigned to the external memory (*assign_LE_tasks_2_ExtMem* + *schedule_reconfigurations*). Figure 88 depicts the first *current_schedule* of the second step of this mapping algorithm.

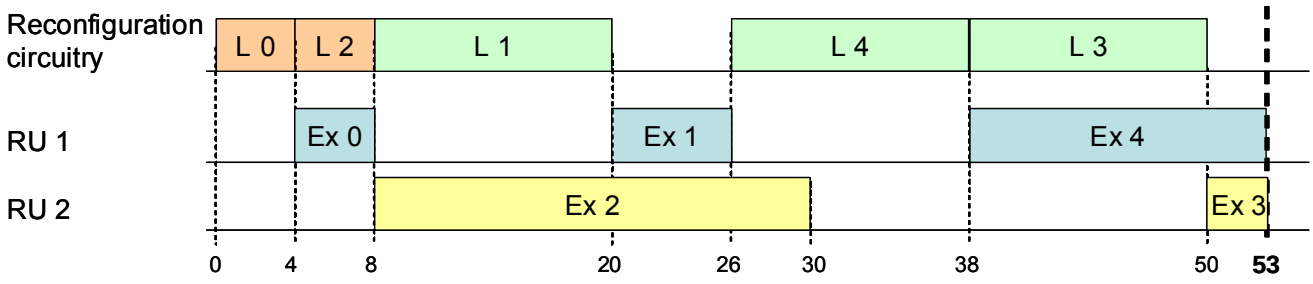


Figure 88. Initial current schedule for the second step of the mapping algorithm (assuming all the tasks are stored in external memory).

Then the mapping algorithm compares the *reference_schedule* (Figure 87) with the *current_schedule* (Figure 88) and identifies which reconfigurations generate delays. In this case the delay is 2 time units, and it has been generated by Task1. Hence, this task is assigned to the LE memory and the function

schedule_reconfigurations is invoked again to generate a new *current_schedule*. Figure 89 presents the resulting *current_schedule* and its execution profile.

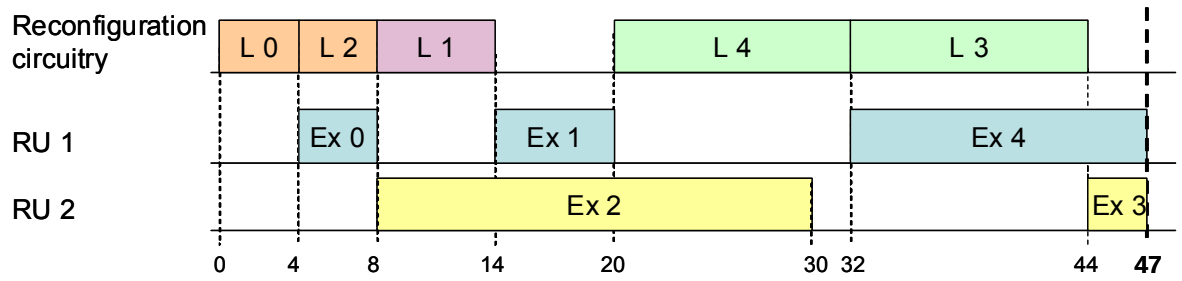


Figure 89. Iteration 1 current schedule for the second step of the mapping algorithm (once task 1 has been moved from the external memory to the LE memory).

As in the previous iterations, the mapping algorithm compares (*compare*) this schedule with the *reference_schedule* and identifies the differences. In this case the execution delay is again 2 time units (*penalty*). Hence, another task is assigned to the LE memory. In this case the task selected is Task 4. After this assignment, the algorithm invokes again the *schedule_reconfigurations* function to generate the new *current_schedule* that is presented in Figure 90.

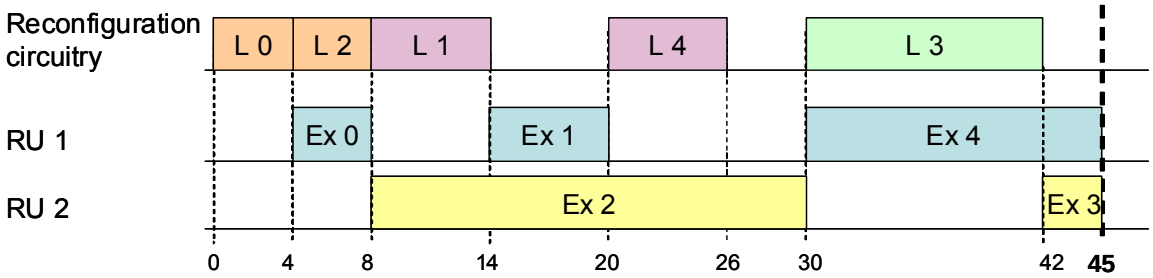


Figure 90. Iteration 2 current schedule for the second step of the mapping algorithm (once task 4 has been moved from the external memory to the LE memory).

This time the comparison between the new *current_schedule* and the *reference_schedule* shows that both of them provide the same execution time.

Hence, the mapping algorithm has reached its final solution allocating the Task 0 and 2 to the HS module; Task 1 and Task 4 to the LE module; and finally Tasks 3 is tagged as “non-cacheable” task and therefore it will be always loaded from the external memory.

If we compare the results obtained by the two mapping algorithms presented in this chapter (one targeted to static systems and the other one oriented to dynamic ones) for the motivational example task-graph, we can see that they have achieved the same solution, although their targets are completely different. The reason is that we are assuming a very demanding scenario in which no extra delays are accepted. This leaves few chances for the second algorithm to move configurations to the external memory. However, normally different results will be obtaining with each algorithm. The next section illustrates this with a more complex example. But before the example we will briefly discuss the impact of the replacement policy on the efficiency of the on-chip memories, and we will propose a simple modification of the well-known LRU replacement policy that can significantly improve the results.

7.4. Improvement of the replacement policy

As we have explained before, our on-chip memory can be seen as an associative cache with two sets: one for the configurations assigned to LE and the other for those assigned to HS. In such a system, the replacement decision can be critical. We have initially selected the well known LRU approach. However, LRU is very sensitive to trashing problems, and for small memories the results can be very

bad. Hence we have slightly modified the LRU policy, significantly improving the results for small memories.

The new replacement policy is quite simple, and the extra clock cycles needed to apply it are negligible when compared with configuration loading latency, and the HW overhead is very small: some registers to store the task-graph ids, a multiplexer, a comparator, and a very simple state machine.

The basic idea is simple: a task of a given task-graph, should not try to replace other tasks of the same graph. When a task of a given graph is going to start its execution, we can split the remaining tasks of that task-graph in two categories. Category 1 includes those tasks of the task-graph that have already started their execution, or even already finished. Since a very recent use of this task has been present, the LRU approach will assign them a high priority. On the contrary, category 2 includes those tasks that have not been executed yet. Since they belong to the task-graph in execution, we know that they are going to be executed soon. However, LRU does not know it and, since their previous execution was further away in the past than the execution of task in category 1, it will possibly select one of them for replacement, and that is exactly what we want to prevent.

The replacement steps are:

1. Apply LRU to select a victim.
2. If the victim belongs to the same task-graph, repeat and select the next victim.

3. If all the victims belong to the same task-graph select the first victim. With our mapping algorithms this should however never happen because our two mapping algorithms never assign more tasks to a memory module than its maximum capacity.

The examples of the following section will illustrate the benefits of this small modification.

7.5. Static and dynamic mapping algorithm comparative behaviour

In this section, we compare the behaviour at run-time of the results obtained using the two proposed mapping algorithms for different execution examples, of both static and dynamic systems. Each mapping approach presented in this chapter takes into account the inherent execution demands of different kinds of systems. With the following analysis, we attempt to show how each of them faces the challenges that arise when no all the tasks executed in the reconfigurable HW can be loaded from the on-chip memory level.

In order to carry out the analysis proposed in this section we have considered two different applications: a sequential version of the JPEG decoder and a sequential version of the MPEG-1 encoder. Figures 13 and 14 depict the initial task-graph schedule for both applications respectively.

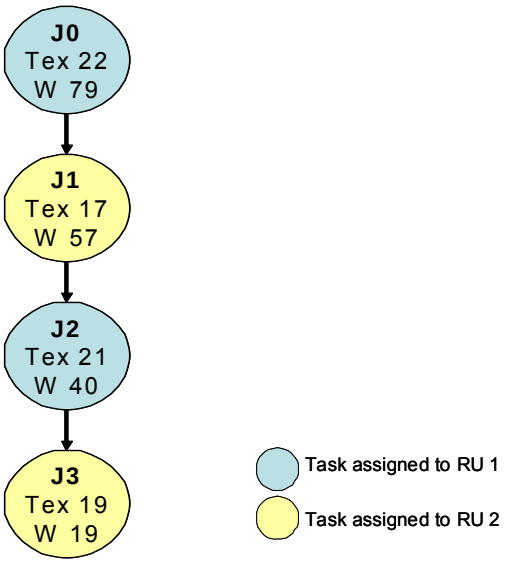


Figure 91. Task-graph for the sequential JPEG.

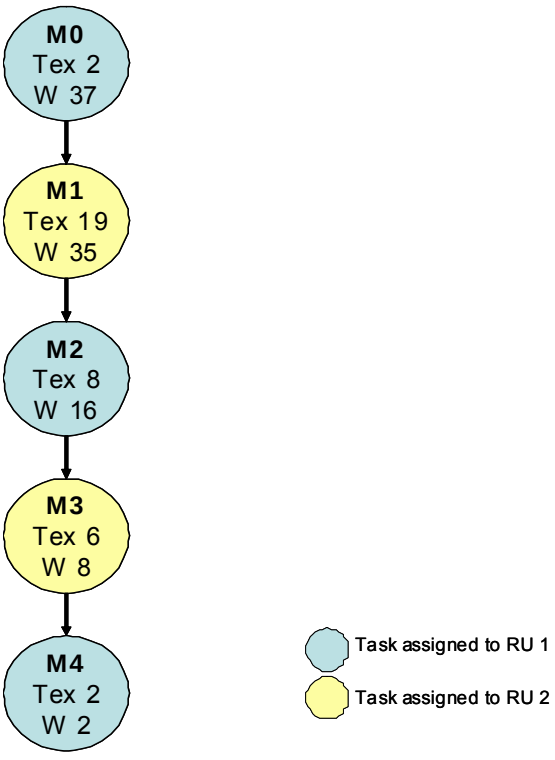


Figure 92. Task-graph for the sequential MPEG-1.

For the execution example of this chapter, we take into account the time and energy consumption due to the configuration movement among all the different levels of the configuration memory hierarchy assuming that all the tasks to be executed are

initially in the external memory. In these examples, we use the same data as in previous ones to characterise the different memories. Table 8 sums up the loading latencies and the energy-consumption per access for each case. For simplicity we consider that an access is an operation that reads or writes a configuration. Hence, reading a configuration from the external memory and storing it in the on-chip LE will consume 4.7 energy units.

Memory Modules	***Load latency (in time units)***	***Energy per access (normalized)***
On-chip HS	4	1
On-chip LE	6	0.7
External Memory	12	4

Table 8. Features of the used memory modules.

In this example, we assume that the system includes two RUs. And that the on-chip HS/LE configuration memory can store four configurations: two in the HS module, and two in the LE module.

Firstly, we will analyse the execution of each graph separately using the solution developed for static systems. The optimal task mapping applying this algorithm for the JPEG task-graph consists in allocating Task 0 and Task 1 into the HS memory, and Task 2 and Task 3 into the LE memory module. In this case, no task has to be loaded into the RUs from the external memory, since, as we have mentioned before, this algorithm attempts to allocate as many tasks as possible into the on-chip memory modules. Hence, in this case, taking into account the features of

the target platform, it is possible to allocate all the tasks of the JPEG task-graph into the on-chip memory modules. For the MPEG-1 task-graph the optimal task mapping applying the algorithm designed for static systems consists in assigning Task 0 and Task 1 to the HS memory, Task 2 and Task 3 to the LE memory module, and tagging Task 4 as “non-cacheable” task, loading it always from the external memory. Figures 15 and 16 present the execution profile of these graphs, if the selected configurations have been preloaded in the on-chip memories.

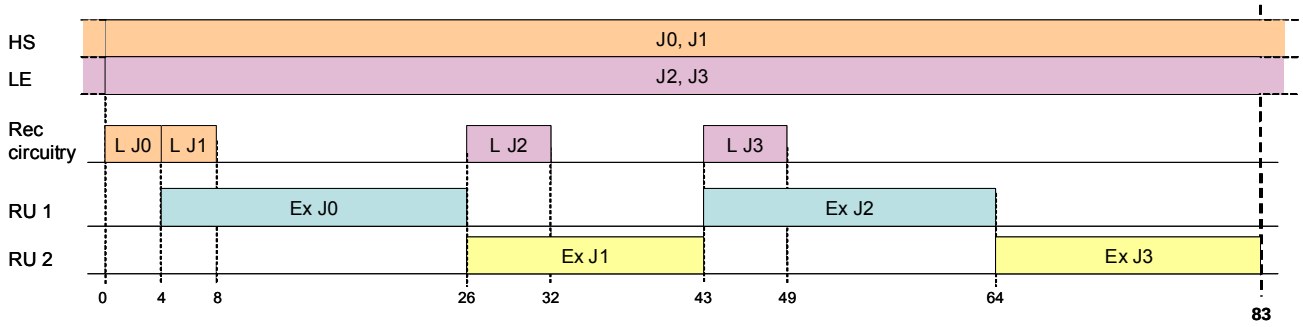


Figure 93. JPEG execution profile.

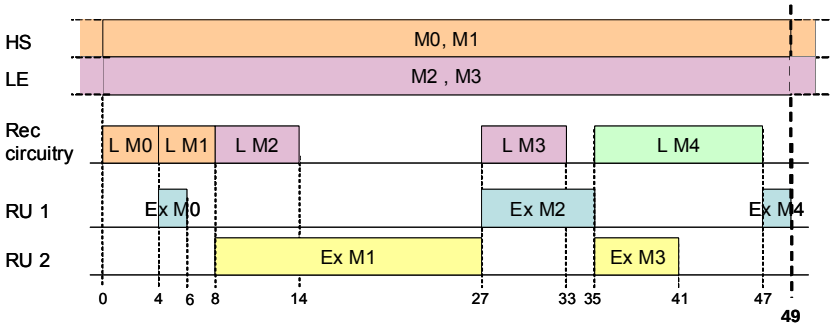


Figure 94. MPEG-1 execution profile.

In both cases the selected solutions achieve the optimal performance for the given platform, and achieve important energy savings when compared with a system that loads everything from the external memory, or a system that only includes a HS module. For the JPEG graph the execution is 83 time units, and the delays due to the

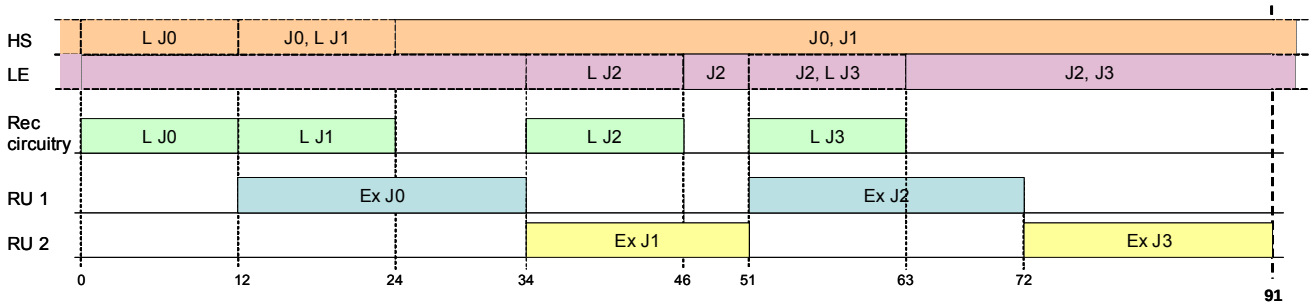
reconfigurations are 4 time units. The MPEG-1 graph execution consumes 49 time units. Of them, 12 units are due to the reconfigurations. From these 12 time units, 6 are due to the load of the configurations from the external memory to the RUs. Since, not enough margin exists to hide the load latency of Task 4. This could only be improved including an LE module with greater capacity.

Regarding the energy overheads due to the configuration memory, for the JPEG task graph the energy consumption will be 3.4 energy units to read the data from the on-chip memory level. And for the MPEG-1 it will be 7.4 energy units: 4 to read the configurations from the external memory, and 3.4 to store four of them in the on-chip memory. If we remove the on-chip level, and load everything from the external memory the energy consumption will be 16 units for the JPEG and 20 units for the MPEG-1.

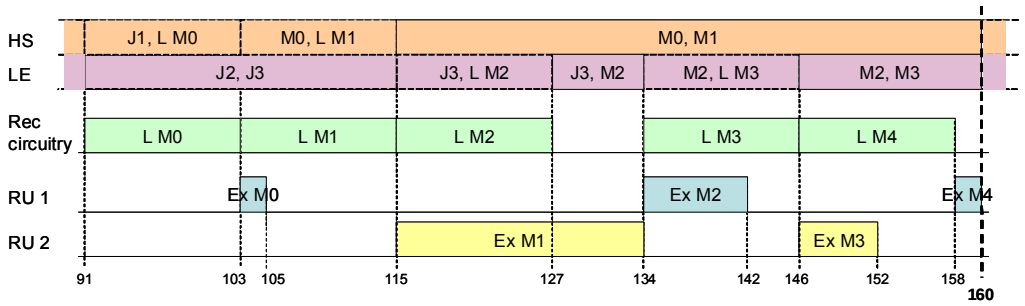
Hence, the solutions obtained with this algorithm are very satisfactory, and it may not be clear why we have also proposed another mapping algorithm. To illustrate this point, we will now assume that instead of executing only one task-graph at a time, the system needs to execute both task-graphs in the same architecture as in the previous example in a loop that includes the JPEG task-graph, followed by the task-graph that represents the MPEG-1.

Figure 95 depicts the execution profile for the first iteration of the combined JPEG, MPEG-1 application, assuming that no tasks have been preloaded in the on-chip memory modules. Hence, during the first iteration, all the configurations have to be loaded into the reconfigurable HW from the external memory. Each time that a configuration is loaded form the external memory, it will be also loaded in the

corresponding on-chip memory module, unless it has been tagged as “non-cacheable”, in order to optimise the execution of the following iterations. In this example we initially assume that our system is applying a simple LRU replacement policy.



a)



b)

Figure 95. First iteration of the JPEG, MPEG-1 application applying the mapping algorithm designed for static systems. a) JPEG b) MPEG-1.

As it can be seen in Figure 95, the total execution time of the first iteration of our application is 160 time units. On the one hand the JPEG graph execution is 91 time units, and the delays due to the reconfigurations are 12 time units. The optimal execution time for this graph is 83 time units that is the execution time obtained when all the configurations can be loaded from a HS memory. Hence, this execution is also 8 times units worse than the optimal one.

On the other hand, the MPEG-1 graph execution consumes 69 time units. Of them, 32 time units are due to the load of the configurations from the external memory to the RUs (and simultaneously into the on-chip memory layer, if the loaded task is a “cacheable” task). In this case the optimal execution time is 43 time units. Hence this execution is 26 units worse than the optimal one.

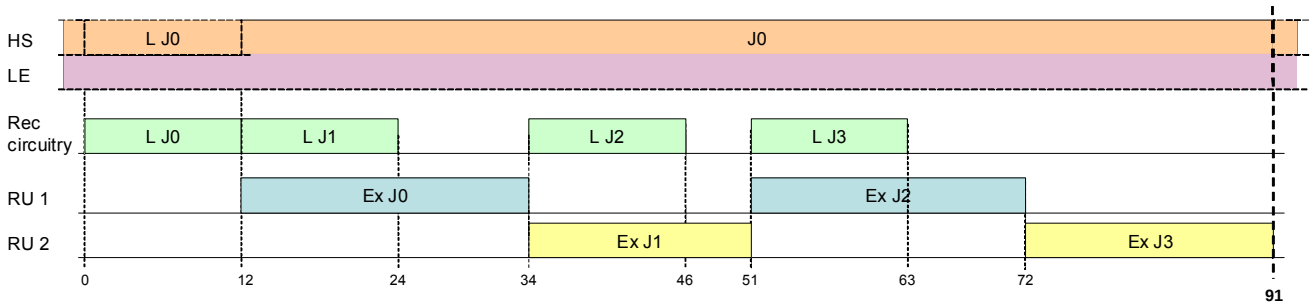
Regarding the energy overheads due to the configuration memory, for the JPEG task graph the energy consumption will be 19.4 energy units: 16 energy units to read the data from the external memory, and 3.4 units to store it in the on-chip configuration memory; and for the MPEG-1 it will be 23.4 energy units: 20 to read the configurations, and 3.4 to store four of them in the on-chip memory. In total, the first iteration of our application consumes 42.8 energy units due to the task configurations movements.

Hence, this approach has consumed even more energy than a solution with no on-chip memory (22% more). Moreover, if we analyse the state of the system at then end of the first iteration it can be seen that the following iterations will have an identical profile as the first one. The reason is that all the configurations that have been loaded in the on-chip modules are going to be replaced before being used. Hence the on-chip modules are wasting energy while introducing no benefits. This is a clear example of a trashing problem. Clearly, this approach leads to a very sub-optimal solution in performance, the same as in a system without on-chip memory level, and to an equally bad solution regarding the energy consumption. The reason is that selecting two local optima usually does not lead to a good solution of the global problem.

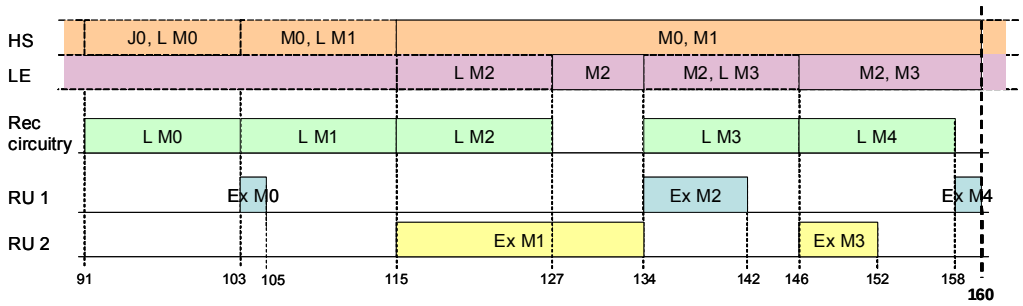
When more than one graph is active in the system (the task mapping is carried out for a dynamic system), our second mapping algorithm can provide better results. In this case, for the JPEG graph, the algorithm will assign Task 0 to the HS on-chip memory module, and will tag Task 1, Task 2 and Task 3 as “non-cacheable” tasks. The task mapping selected will be the same as in the previous case, since no margin exists to move more configurations to the external memory. Hence, the mapping for the MPEG-1 graph consists again in assigning Task 0 and Task 1 to the HS memory, Task 2 and Task 3 to the LE memory module, and tagged Task 4 as “non-cacheable” task.

Figure 96 presents the execution profile for the first iteration of the combined execution of the JPEG and MPEG-1 application assuming that no configuration has been preloaded and for the same platform as in the previous cases, but applying the mapping approach designed for dynamic systems. We also assume for this example that a simple LRU replacement policy is applied.

As it can be seen in Figure 96, the total execution time of the first iteration of our application is 160 time units, i.e. the same as when we were applying the previous mapping. Regarding the energy overheads due to the configuration memory, for the JPEG task graph the energy consumption will be this time 17 energy units: 16 energy units to read the data from the external memory, and 1 unit to store it in the on-chip configuration memory; and for the MPEG-1 it will the same as the previous example, i.e., 23.4 energy units: 20 to read the configurations, and 3.4 to store four of them in the on-chip memory. This time, the total energy consumption is 40.4 energy units, which is 2.4 units less than for the previous example.



a)



b)

Figure 96. First iteration of the JPEG, MPEG-1 application applying the mapping algorithm designed for dynamic systems with the simple LRU. a) JPEG b) MPEG-1.

Hence, the first iteration has been very similar, which is logical since all the configurations have been fetched form the external memory. However, if we analyse the state of the system at then end of the first iteration, it can be seen that the following iterations are going to have a different profile since it will be possible to fetch some of the configurations form the on-chip modules.

Figure 97 presents the execution profile for the second iteration of the combined JPEG, MPEG-1 application for this example.

As it can be seen in Figure 97, the total execution time of the second iteration of our application is 156 time units. Thus, it is 4 time units less than the execution time of the first iteration. On the one hand the JPEG graph execution is 91 time units, the same as for the first iteration, as then, 8 times units worse than the optimal one.

On the other hand, the MPEG-1 graph execution consumes 65 time units, and 28 of these units are due to configurations fetched from the external memory to the RUs (4 time units less than for the first iteration). The reason is that MPEG-1 Task 2 and MPEG-1 Task 3 have been fetched from the LE module. However, the performance is still far from the optimum.

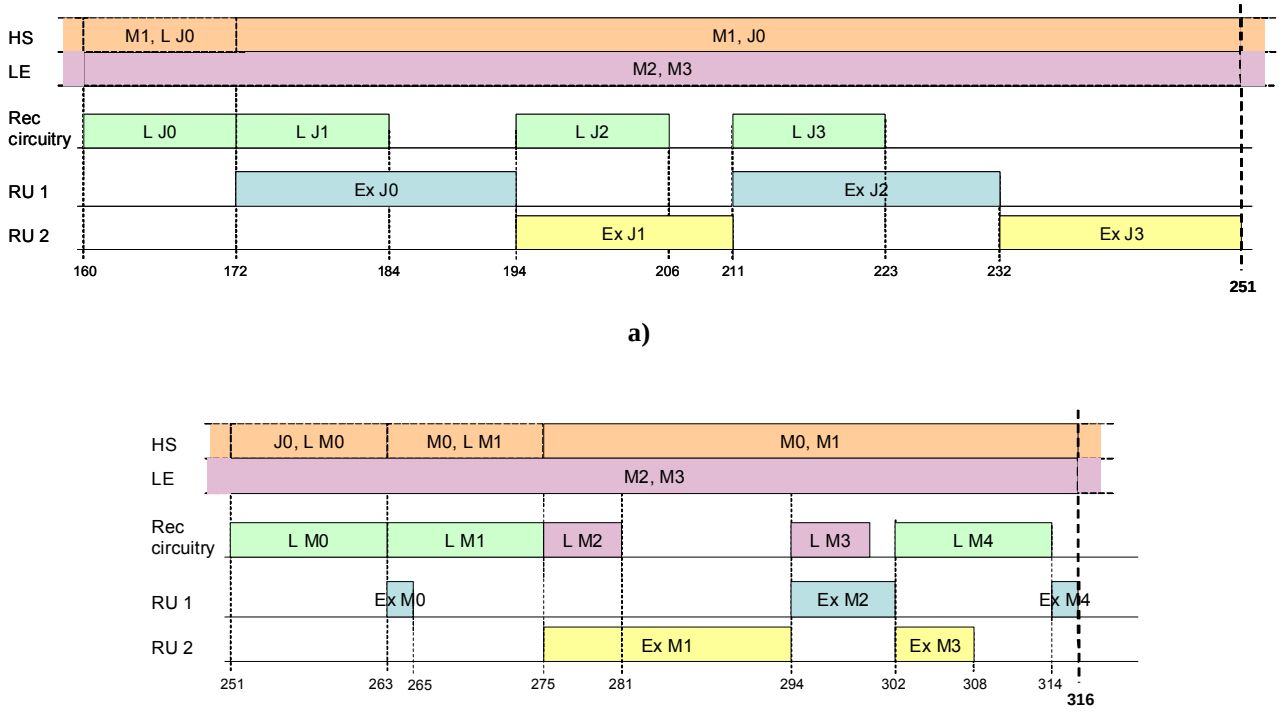


Figure 97. Second iteration of the JPEG, MPEG-1 application applying the mapping algorithm designed for dynamic systems with the simple LRU. a) JPEG b) MPEG-1.

Regarding the energy overheads due to the configuration memory, for the JPEG task graph the energy consumption will be also the same as in the first iteration, 17 energy units. Since, the only task tagged as “cacheable” from the on-chip memory level (Task 0), has been replaced before using it. Nevertheless, for the MPEG-1 some energy-savings are achieved consuming 15.4 energy units instead of

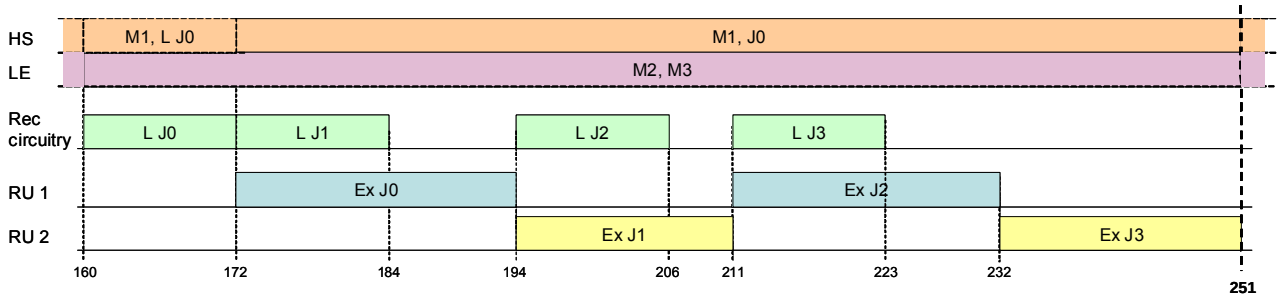
23,4. Hence, for the second iteration this approach consumes 32.4 energy units, 3.6 less energy units than a solution with no on-chip memory.

If we analyse the state of the system at then end of the second iteration, it can be seen that the following iterations are going to be similar as the second one. Indeed, at the end of it the configurations stored in the on-chip memory level are the same as at the end of the first iteration.

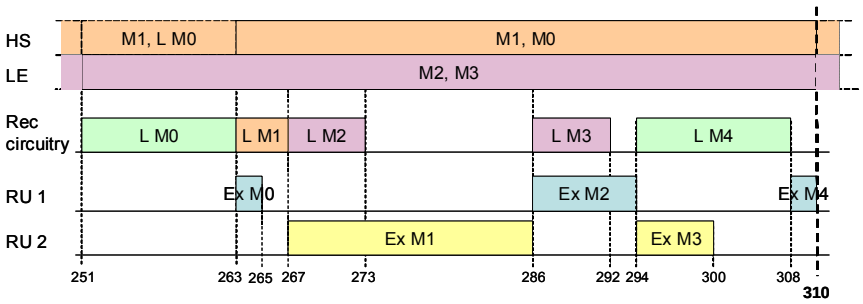
Therefore, this approach has improved the results of the previous one both in terms of energy consumption and of performance. However, still better results can be obtained applying a smarter replacement policy. We will carry out the same experiment but applying the extension of the LRU replacement policy that we have described previously. The first iteration will be similar since nothing was replaced. Figure 98 depicts the execution profile for the second iteration of the combined JPEG, MPEG-1 application, using the proposed modified LRU replacement policy.

As it can be seen in Figure 98, the total execution time of the second iteration of our application is this time 148 time units. Thus, this replacement technique has improved the results with 6 time units when compared with the LRU case, and 12 time units less when compared with the first iteration. However, we have still overheads even with this replacement policy that in this simple example is making the optimal replacement decisions for the given mappings. Indeed, still a thrashing problem exists, since these mappings require storing three tasks in the HS module, and space is only available for two. The improvement has come because now one of the MPEG-1 tasks can be loaded from the HS memory. With the LRU replacement policy the trashing was affecting to all the tasks assigned to HS, and with the

improved replacement policy, one of the configurations is not affected. However, for the given space, we are now at the global optimum for the performance.



a)



b)

Figure 98. Second iteration of the JPEG, MPEG-1 application applying the mapping algorithm designed for dynamic systems with the modified LRU. a) JPEG b) MPEG-1.

Regarding the energy overheads due to the configuration memory, for the JPEG task-graph the energy consumption will be also the same as in the first iteration and in the previous examples of this section, 17 energy units. However, for the MPEG-1 the energy overhead has been reduced to 11.4 units, since, one more task is fetched from the on-chip memory. Hence, for the second iteration, the energy consumed due to the configuration memory hierarchy will be 28.4 energy units, 7.6 energy units less energy than a solution with no on-chip memory, and 20,4 units less than a system that applies the mapping proposed by the first algorithm., that was in

fact the optimal solution regarding energy for the given performance constraint and configuration size.

As in the previous example, due to the state of the on-chip memory modules, the following iterations are going to have the same profile as the second one. Hence, subsequent iterations will have the same performance and consume the same energy as the second one.

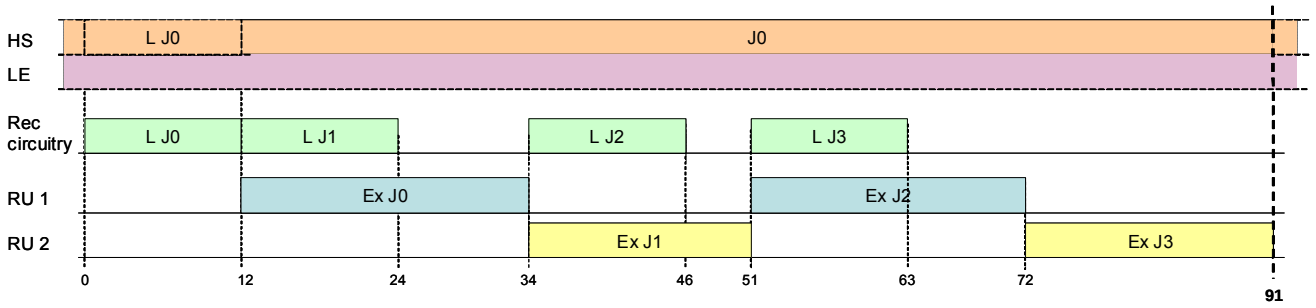
Therefore, with this modification in the replacement policy, this system starts to be significantly better than a similar system with no on-chip modules. In particular, it is 8% better in performance and 21% better in energy consumption.

In order to look for even better solutions in performance and energy consumption, we should resize our on-chip modules. The problem is clear: we have two active task-graphs that need to store 5 configurations in the on-chip memory in order to achieve the optimal performance. Hence, if the on-chip modules can only store 4 configurations, the results will always be suboptimal.

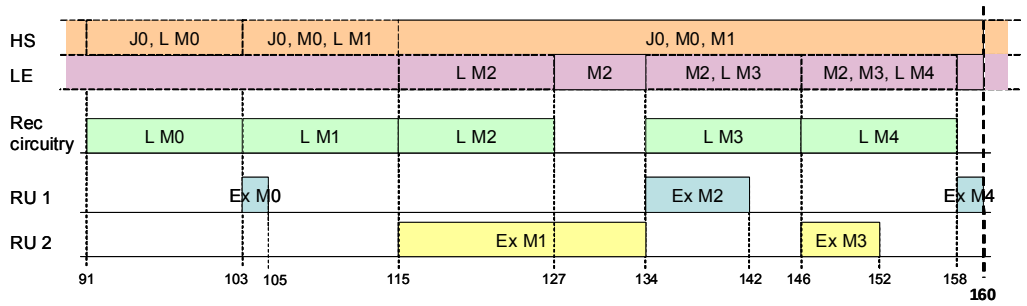
We will now resize the on-chip memory modules in order to provide the needed storage. We will assume that six configurations can be stored, three in each module.

For the new target platform, the mapping algorithm is applied again separately for each task-graph of the application. The mapping solution for the JPEG graph is the same as for the previous architecture. But it will be different for the MPEG-1, since the previous mapping was not achieving the optimal results due to the lack of space in the LE memory. The new mapping for the MPEG-1 graph assigns Task 0 and Task 1 to the HS memory, and Task 2, Task 3 and Task 4 to the LE memory.

Figure 99 depicts the execution profile for the first iteration of the JPEG, MPEG-1 applying the mapping approach designed for dynamic systems application, and assuming it is executed in a system, that as in the previous example, includes two RUs, but that the on-chip HS/LE configuration memory can store now six configurations: three in the HS module, and three in the LE module. We also assume for this example that the system is using the modified LRU replacement policy proposed earlier in this chapter.



a)

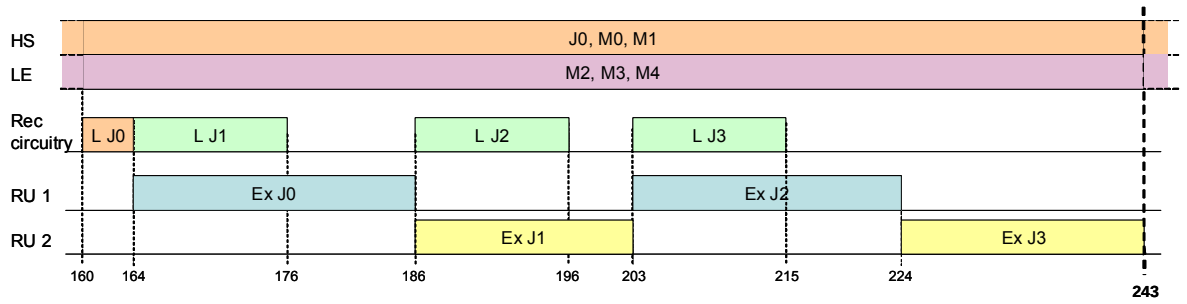


b)

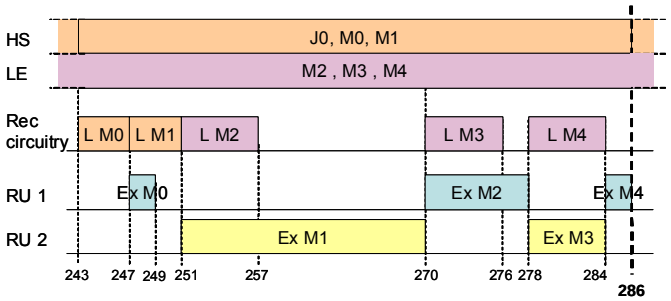
Figure 99. First iteration of the JPEG, MPEG-1 application applying the mapping algorithm designed for dynamic systems. a) JPEG b) MPEG-1.

Although the execution profile is different from the previous examples, due to the extra movement of MPEG-1 Task 4 between the external memory and the on-chip memory level, the total execution time of the first iteration of our application is the same as before, 160 time units, because in the first iteration configurations are

always loaded from the external memory. Regarding energy, the MPEG-1 will consume slightly more (0.7 energy units) than in the previous case. However, both energy and performance will greatly improve in the following iterations as is depicted in Figure 100.



a)



b)

Figure 100. Second iteration of the JPEG, MPEG-1 application applying the mapping algorithm designed for dynamic systems with the modified LRU. a) JPEG b) MPEG-1.

The total execution time of the second iteration is just 126 time units. i.e. 34 time units less than the execution time of the first iteration. On the one hand, the JPEG graph execution is 83 time units, the same as the optimal one. Indeed, this time Task 0, which is the only configuration task load of the JPEG graph that entails a performance delay, can be loaded from the HS on-chip memory module instead of from the external memory. On the other hand, the MPEG-1 graph execution lasts 43 time units, also the same as the optimal execution time.

With regard to the energy overheads due to the configuration memory, this system achieves important energy-savings in the second iteration. In the JPEG task graph the energy consumption for the second iteration of the application will be 8 units less than for the first iteration: 13 energy units. And for the MPEG-1 it will be much smaller than for the previous iteration: namely 4.1 energy units (20.3 energy units less than the first iteration). This has been achieved since all the configurations are now loaded from the on-chip memory module. Hence, for the second iteration this approach has consumed only 17.1 energy units, which is18.9 units less energy than a solution with no on-chip memory.

If we analyse the state of the system at the end of the second iteration, it can be seen that the following iterations are going to have the same profile as it. Since, at the end of the second iteration the configurations stored in the on-chip memory level are the same that at the end of the first one. Hence, apart from the first iteration, the mapping algorithm for dynamic systems in the redesigned platform obtains, for the second and subsequent iterations, the optimal solution in performance, which is 21% better than the solution for the system without on-chip memories. And at the same time it achieves important energy savings: it saves 53% of the energy consumed by the configuration movements by a system without the on-chip level.

Hence, for systems with only one task-graph the first algorithm should be used, whereas for more than one task-graph the second one will most probably provide better results, reducing the possibilities of a trashing problem. In addition, if we want to achieve optimal results in performance, the number of task-graphs active at the

same time should be limited taking into account the size of the on-chip modules and the number of configurations assigned to the HS and LE modules for each graph.

7.6. Experimental Results

To demonstrate our new modules, we have integrated them into the scheduling environment presented in chapter 5, and we have carried out two different set of representative experiments, one for a fine-grain reconfigurable platform and another for a coarse-grain platform, for the two mapping algorithms proposed in this chapter. To carry out the experiments we have used the same simulation environment that has been described in the previous chapters.

In the first experiment we have evaluated the mapping algorithms with a set of multimedia applications assuming that all the tasks are executed in reconfigurable units and that the configurations can be fetched from an on-chip HS memory, an on-chip LE memory or an external one. The applications are the same as in chapter 6, i.e., a sequential and a parallel version of the JPEG decoder, and the MPEG-1 encoder, and a Pattern Recognition application that applies the Hough transform over a matrix of pixels in order to identify geometrical figures.

Table 9 presents the data regarding latency and energy consumption for the three different options available to fetch a task. As we have explained previously, this data has been obtained using real data from ST microelectronics, and estimated data obtained from CACTI.

Memory Modules	*Reconfiguration latency*	*Normalized energy consumption*
On-chip HS	4 ms	1
On-chip LE	6 ms	0,7
External memory	12 ms	4

Table 9. Features of the fine-grain platform memory modules.

In Table 10, the features of this set of applications are presented. **Tasks** is the number of tasks of each task-graph. **Ideal ex. time** is the average execution time of each application without reconfiguration overhead. The three columns under the headline **Execution time overhead**, **Ext**, **HS** and **LE** are the execution-time reconfiguration overhead when all the reconfigurations are loaded from external memory, from the HS on-chip-memory module and from the LE one, respectively.

Application	*Tasks*	*Ideal ex. time*	*Execution time overhead*		
			Ext	*HS*	*LE*
Pattern Rec.	6	94 ms	+13%	+4%	+6%
JPEG dec.	4	79 ms	+15%	+5%	+8%
Parallel JPEG	8	54 ms	+72%	+7%	+26%
MPEG enc.	5	37 ms	+76%	+16%	+27%
Average			**+44%**	**+8%**	**+17%**

Table 10. Features of the fine-grain task-graphs.

Tables 11 and 12 present the results of our fine-grain experiments applying our mapping approach for static systems and for dynamic systems, respectively.

Application	Tasks			Energy rec. (normalized to always Ext)	
	Ext	HS	LE	1st it.	Remaining it.
Pattern Rec.	0	3	3	1,21	0,21
JPEG dec.	0	1	3	1,19	0,19
Parallel JPEG	2	3	3	1,16	0,41
MPEG encoder	1	2	2	1,17	0,37
Average	**13%**	**39%**	**48%**	**1,18**	**0,30**

Table 11. Experimental results for fine-grain reconfigurable HW applying our mapping approach for static systems.

Application	Tasks			Energy rec. (normalized to always Ext)	
	Ext	HS	LE	1st it.	Remaining it.
Pattern Rec.	5	1	0	1,04	0,88
JPEG dec.	3	1	0	1,06	0,81
Parallel JPEG	2	3	3	1,16	0,41
MPEG enc.	1	2	2	1,17	0,37
Average	**48%**	**30%**	**22%**	**1,11**	**0,62**

Table 12. Experimental results for fine-grain reconfigurable HW applying our mapping approach for dynamic systems.

In this Table, the three columns under the headline **Tasks**: **Ext**, **HS** and **LE**, represented the selected mapping. The energy consumption due to the load of configurations into the reconfigurable HW is present in Tables 4 and 5 normalized to the configuration energy-consumption when all tasks are always loaded from the external memory.

It is important to remark that in all the cases both algorithms obtain the optimal performance for all the iterations but the first one, while assigning only 39% and 30% of the configurations to the HS memory.

For both mapping approaches the energy consumption of loading configurations in the reconfigurable HW during the first iteration of the task-graph execution is always a bit greater than the energy consumption when all tasks are loaded from external memory (i.e. it is greater than one). The reason is that we are not assuming that the configurations assigned to the on-chip memories have been preloaded to the on-chip modules. Hence the first time they are fetched from the external memories and stored in the on-chip modules simultaneously to the run-time reconfiguration, introducing and energy penalization (18% for the first algorithm and 11% for the second one). This initial penalisation is greater in the first case since more tasks have been assigned to the on-chip modules.

However, as soon as the task graph is executed several times, this initial energy penalisation will be quickly compensated with the energy savings obtained in the subsequent iterations. In this case the first algorithm produces the better results eliminating 70% of the energy consumption due to the reconfiguration. The second algorithm also achieves important savings (almost 40%). The results are worse in this latter case because this algorithm attempts to keep as many configurations as possible in the external memory in order to prevent possible trashing problems. Table 13 shows the comparison between the results obtained applying both proposed algorithms.

As can be seen in the table, these algorithms provide a trade-off between energy efficiency and the pressure on the on-chip memories. It will be the designer decision to choose the algorithm that better fits to his system, taking into account the expected run-time pressure in the on-chip memory modules.

Application	*Tasks assigned to on-chip mem. level*		*Energy rec. (2nd & subsequent it.)*	
	Static systems algorithm	***Dynamic systems algorithm***	***Static systems algorithm***	***Dynamic systems algorithm***
Pattern Rec.	100%	17%	5,1	21
JPEG dec.	100%	25%	3,1	13
Parallel JPEG	75%	75%	13,1	13,1
MPEG enc.	80%	80%	7,4	7,4
Average	**89%**	**49%**	**7,2**	**13,6**

Table 13. Experimental results comparison of static and dynamic mapping algorithms for fine-grain reconfigurable HW.

As we can see in Table 13 the mapping algorithm for dynamic systems only assigns 49% of the configurations to the on-chip memory layer, since it identifies that the remaining ones can be fetched from the external memory without degrading the performance. This reduction of the pressure on the on-chip level comes at a cost of an energy penalisation when compared to the approach developed for static systems. However, these numbers have been obtained assuming that no thrashing problems exist. If this assumption is not true, the energy numbers of the approach developed for static systems can be worse, as it was explained in the detailed example of the previous section.

If we focus on the result obtained for the parallel version of the JPEG decoder, and for the MPEG-1 encoder, we can see that both mapping algorithms have obtained the same results. The reason is simple; the first objective of both algorithms is to meet the performance constraint. In these cases the constraints are very tight

and, in order to meet them, four tasks must be fetched from the on-chip modules. If the performance constraints for these two applications are relaxed, the mapping algorithm for dynamic systems could tag some tasks as “non-cacheable” tasks, achieving similar results than those obtained for the sequential version of the JPEG decoder, and the Pattern Recognition applications.

In order to evaluate our approaches for coarse-grain platforms we have used, as in chapter 6, the coarse-grain simulation environment CRISP presented in chapter 2. As we have mentioned before, this platform is also very similar to the domain-specific VLIW architectures that become very popular for low-energy high-performance wireless and multi-media application kernels. In this environment the designer can define the coarse-grain architecture and the memory hierarchy. In our case, we have defined an architecture with two different loop buffers from where instructions are loaded into the coarse-grain units. The configurations are loaded into the loop buffers either from the external memory of the systems, or from one of the two on-chip memory modules (a HS module and a LE module). Table 14 presents the latency of a reconfiguration for these three options, and the normalised energy consumption numbers. Hence, in this experiment the configuration latency is drastically smaller than in the previous one. Therefore, we will apply our mapping technique for a smaller task granularity. As in chapter 6, for our experiments with the defined coarse-grain architecture, we have selected a set of DSP benchmarks developed by Texas Instruments [TeIn09].

The details of the DSP benchmarks selected task-graph are depicted in Table 15. As for fine-grain applications, **Tasks** is the number of tasks of each task-graph.

Ideal ex. time is the average execution time of each application without reconfiguration overhead. The **Execution time overhead**, **Ext**, **HS** and **LE** are the overhead due to the reconfigurations when all the configurations are loaded from the external memory, the HS module, and the LE module, respectively.

Memory Modules	***Reconfiguration latency***	***Normalized energy consumption***
On-chip HS	6 µs	1
On-chip LE	9 µs	0,7
External memory	18 µs	4

Table 14. Features of the coarse-grain platform memory modules.

Application	***Tasks***	***Ideal ex. time***	***Execution time overhead***		
			Ext	***HS***	***LE***
DSP_dot_prod	3	32 µs	+71%	+19%	+28%
DSP_vec_sumq	2	32 µs	+56%	+19%	+28%
DSP_q15_tofl	3	5645 µs	+0%	+0%	+0%
DSP_neg32	3	109 µs	+18%	+6%	+8%
DSP_min_val	3	114 µs	+18%	+5%	+8%
DSP_dotp_sqr	3	32 µs	+70%	+19%	+28%
DSP_blkmove	2	9 µs	+391%	+125%	+191%
Average			**89%**	**27%**	**42%**

Table 15. Features of the coarse-grain task-graphs.

For this task granularity the impact of the reconfiguration latency is even greater than in the previous experiment. For example due to the small granularity of the DSP_blkmove graph the impact of the reconfiguration latency over the whole

application performance is in all cases greater than 100%. On the contrary, for the DSP_q15_tofl application the reconfiguration overhead can be neglected since it does not influence the task-graph performance.

Application	***Tasks***			***Energy rec. (normalized to always Ext)***	
	Ext	***HS***	***LE***	***1st it.***	***Remaining it.***
DSP_dot_prod	0	1	2	1,20	0,20
DSP_vec_sumq	0	1	1	1,21	0,21
DSP_q15_tofl	0	1	2	1,20	0,20
DSP_neg32	0	1	2	1,20	0,20
DSP_min_val	0	1	2	1,20	0,20
DSP_dotp_sqr	0	1	2	1,20	0,20
DSP_blkmove	0	2	0	1,25	0,25
Average	**0%**	**42%**	**58%**	**1,21**	**0,21**

Table 16. Experimental results for coarse-grain applying our mapping approach for static systems.

Application	***Tasks***			***Energy rec. (normalized to always Ext)***	
	Ext	***HS***	***LE***	***1st it.***	***Remaining it.***
DSP_dot_prod	0	1	2	1,20	0,20
DSP_vec_sumq	1	1	0	1,13	0,63
DSP_q15_tofl	1	1	1	1,14	0,48
DSP_neg32	1	1	1	1,14	0,48
DSP_min_val	1	1	1	1,14	0,48
DSP_dotp_sqr	0	1	2	1,20	0,20
DSP_blkmove	0	2	0	1,25	0,25
Average	**21%**	**42%**	**37%**	**1,17**	**0,39**

Table 17. Experimental results for coarse-grain applying our mapping approach for dynamic systems.

Tables 9 and 10 present the results of our coarse-grain experiments applying our mapping approach for static systems and for dynamic systems, respectively.

As for fine-grain experiments, the three columns under the headline **Tasks** present the selected mapping, and the following two columns present the energy reconfiguration overhead, both for the first iteration, and the subsequent ones, normalized using the energy overhead obtained when all the configurations are fetched form the external memory.

As for fine-grain experiments, our two mapping algorithms introduce an energy penalization in the first iteration, but they achieve very important energy-consumption reductions in the remaining iterations.

Application	***Tasks assigned to on-chip mem. level***		***Energy rec. (2nd & subsequent it.)***	
	Static systems algorithm	***Dynamic systems algorithm***	***Static systems algorithm***	***Dynamic systems algorithm***
DSP_dot_prod	100%	100%	2,4	2,4
DSP_vec_sumq	100%	50%	1,7	5,0
DSP_q15_tofl	100%	67%	2,4	5,7
DSP_neg32	100%	67%	2,4	5,7
DSP_min_val	100%	67%	2,4	5,7
DSP_dotp_sqr	100%	100%	2,4	2,4
DSP_blkmove	100%	100%	2,0	2,0
Average	**100%**	**79%**	**2,2**	**4,1**

Table 18. Experimental results comparison of static and dynamic mapping algorithms for coarse-grain.

As in the previous experiment, these two algorithms provide a trade-off for the system designer as can be seen in Table 18. In this case the final mappings are more similar, since the execution time constraints were very tight, we are always trying to

achieve the same performance than a system that uses only HS memories) hence the algorithm developed for dynamic systems can only assign 21% of the tasks to the external memory.

Chapter 8:
Conclusions

The reconfiguration latency, and the energy needed to carry out the reconfiguration are two of the most important drawbacks of current reconfigurable platforms. Both drawbacks are present in coarse-grain and fine-grain reconfigurable platforms, but their effects are especially significant in the last one.

However, the ability of partial reconfiguration at run-time is really useful for highly-dynamic applications, since it provides adaptability for the unpredictable events and high performance to implement design taking advantage of the inherent parallelism of each task. Besides, in the embedded system environment the reconfigurable resources are even more attractive, since they can be reused to execute different applications, which entail a minimization of the system area.

Right now, the presented approaches that attempt to minimize the run-time partial reconfiguration overheads only tackle the reconfiguration time overhead, neglecting the energy consumption due to the reconfiguration. With the work developed in this thesis, we try to demonstrate that with the suitable support, it is possible to reduce both overheads, achieving the required performance while minimizing the reconfiguration energy overhead.

In our approach we assume that applications are compose by a set of task-graphs that can be modelled as Direct Acyclic Graphs (DAGs). Each node of these graphs represents a computational task, i.e. a piece of code with enough entity to be separately assigned to a certain processing element. Hence, these tasks are the basic scheduling unit.

Tasks are loaded in the Reconfigurable Units (RU) using partial reconfiguration at run-time. In order to provide the needed communication support for these tasks assigned we assume that each RU is wrapped with a fixed communication interface that provides support to carry out inter-tasks using message passing primitives. Beside, these interfaces must provide the basic functionality needed for the scheduler and for the operative system to manage the tasks execution in these units [NCVV03].

In order to manage the tasks execution in the different processing elements, we have used a scheduler designed for conventional multiprocessor systems, and we have extended it to support reconfigurable resources. The initial scheduler generates an initial schedule that neglects the reconfiguration overheads due to the reconfiguration. Afterwards, some techniques are applied to optimise the reconfigurations, namely: configuration reuse, prefetch and replacement.

The aim of the reuse module is to identify, for a given schedule, which configurations can be reused from a previous execution. The prefetch technique attempts to load configurations in advance in order to hide the loading latency. Finally, the replacement technique attempts to optimise the performance and the reuse of configurations selecting the victim configuration that is replaced when a new task must be loaded. This replacement policy takes into account the information of the current graph, the previous events, and specific information assigned to each task at design-time, in order to optimise its decisions.

All these previous modules were developed for a simple configuration memory hierarchy where all the configurations were fetched from an external memory and directly loaded in the reconfigurable units. However, in order to efficiently deal with HW multi-tasking a more complex hierarchy for configurations is needed. In this thesis we propose a two-level approach. The upper level is the off-chip external memory, whereas the lower level is a heterogeneous on-chip configuration memory layer composed of two memory modules: one optimised for high-speed, and another optimised for low-energy. Hence, the configuration memory hierarchy proposed in this thesis provides fast reconfiguration, and at the same time the possibility of achieving energy savings whenever this speed is not needed. In addition the proposed techniques can be easily extended for systems with more than two levels.

In order to optimise the potential advantages of this heterogeneous memory hierarchy, we have developed different mapping algorithms that have been included in the scheduling system in order to take as much advantages as possible of run-time. Our systematic mapping algorithms analyse at design-time the features of the

graphs that compose an application and interacts with the prefetch module. The mapping algorithm must decide whether configurations should be stored in the low-energy, in the high-speed memory, or in the external memory if it is necessary, in order to load them into the reconfigurable resources. Storing a configuration in the low-energy memory reduces the energy reconfiguration overhead but at the cost of a possible increase in the execution-time. Meanwhile, to store a configuration in the external memory not only entails higher reconfiguration latency, but also increase the energy needed to carry out the required reconfiguration. However, in some cases it is needed to load configurations from the external memory level in order to avoid thrashing problems in the on-chip level.

The goal of our mapping algorithms is to identify a partition of the configurations that minimise the reconfiguration energy overhead while achieving the requested performance. It is very important to identify which are those tasks that have a high impact in the whole system performance, and assign them to the high-speed memory whereas the remaining tasks are stored on the low-energy memory, or on the external memory regarding the execution system features.

We have developed three different mapping algorithms. The firs one (presented in Chapter 6) assumes that all the configurations corresponding to the task in execution can be pre-stored in the on-chip level. The basic idea of this mapping algorithm is to identify a partition of the tasks that minimises the reconfiguration energy overhead without introducing any performance degradation. To accomplish this, the mapping algorithm identifies those tasks which load latencies cannot be hidden and maps them to the high-speed memory in order to minimise the

reconfiguration overhead. The remaining tasks are mapped to the low-energy memory. This is the optimal case, both for performance and for energy-consumption. However, on-chip resources are frequently very limited in embedded systems; and if the number and size of the configurations exceeds the capacity of the on-chip configuration memory this approach can lead to a suboptimal execution results due to the configuration thrashing effect. Configuration thrashing occurs when configurations are fetched and replaced systematically on the on-chip memory, and it is a well-known memory management problem.

To face this problem, we have extended the previous algorithm establishing restrictions at the task-graph level in order to guarantee that the configurations assigned by each graph to the on-chip modules do not exceed their capacity. This new mapping algorithm shares the objective of the previous one: obtain a task partition that provides the best possible performance, while reducing as much as possible the energy consumption. The main difference is that this algorithm takes into account the size of the on-chip modules. Since the on-chip memory modules are typically both better in performance and energy consumption, this algorithm will always attempt to maximise the use of the on-chip memory layer, assigning to it as many configurations as possible. If not enough space is available for all the configurations in the on-chip memory level, the mapping algorithm will assign some of them to the external memory in order to avoid a configuration trashing problem. This extension solves the trashing problems as long as only one task-graph is being executed at a given time. However, it can still lead to trashing problems when it is applied to dynamic systems that support the interleaved execution of several graphs.

To solve this problem, we have developed another mapping algorithm targeting dynamic systems where different task-graphs will be executed several times, and it is not known which graphs will be active concurrently. Since, for these systems we do not know the actual running conditions we cannot look for an optimal solution. Hence, in this case the mapping algorithm looks for the solution that meets a given performance constraint, as long as no run-time conflicts are present, while generating the minimum pressure on the on-chip configuration memory. In addition, it attempts to reduce the energy consumption, mapping conveniently those tasks assigned to the on-chip memory layer. In this case, the mapping algorithm does not only optimise the execution of a single task-graph, but also attempt to reduce possible conflicts in the on-chip memory layer, since these conflicts may generate both important delays in the execution and energy penalisations. Basically, each graph will be analysed separately at design-time to identify those configurations that can be loaded from an external memory without introducing a performance degradation in the whole task-graph execution, or those that will introduce a performance degradation that can be accepted by the system. Once these configurations are identified, they are tagged as “non-cacheable” tasks in order to reduce the pressure on the on-chip configuration memory.

To demonstrate our proposals, we have tested them with two sets of task graphs: one for a fine-grain platform, and another for a coarse-grain platform. The results demonstrate that, with the appropriate mapping algorithm, a hybrid low-energy high-speed on-chip configuration memory achieves the optimal performance while drastically reducing the energy consumption due to the reconfigurations when

compare with a system that stores all the configurations in an external memory. Moreover, our approach achieves the same performance than a system that only uses high-speed modules, while mapping on average half of the tasks in the low-energy module, achieving significant energy reductions.

Regarding the two mapping algorithms presented in chapter 7, it is important to remark that none of them is better than the other, but they provide a trade-off between the memory pressure in the on-chip modules and the energy consumption due to the reconfigurations. The first algorithm minimise the energy consumption assigning as many configurations as possible to the on-chip modules. If only one task graph is active this would be the optimal solution. However, if several task-graphs are competing for the resources, the trashing effect may drastically degrade the performance and increase the energy consumption. In those cases the second algorithm should be used, since it provides the same performance, but assigning as many configurations as possible to the external memory. This approach reduces the on-chip memory pressure at the cost of some energy overhead. Hence the designer should select the appropriated algorithm for each run-time situation.

Clearly, the benefits of our approach depend on the granularity of the tasks and on the configuration latency. If for some given values the reconfiguration overhead is not significant, our mapping technique will not be needed. However, as it can be seen in our experiments, in many cases the reconfiguration overhead has a significant impact both on energy consumption and performance. In these situations our approach achieves relevant energy savings while achieving the requested performance.

Chapter 9:
Future Work

As future work, we are currently working on developing an implementation of the on-chip memory module for FPGAs. The idea is to design a controller that can interact both with the internal FPGA memory resources and with the on-chip reconfiguration circuitry. The configurations would be stored in the on-chip Block-RAMs and the controller will manage these memory resources preventing the fragmentation problem, and applying the replacement policy described in chapter 7.

The basic blocks of this prototype are depicted in Figure 101. As can be seen in the figure, the memory resources will be partitioned in memory blocks of a fixed size.

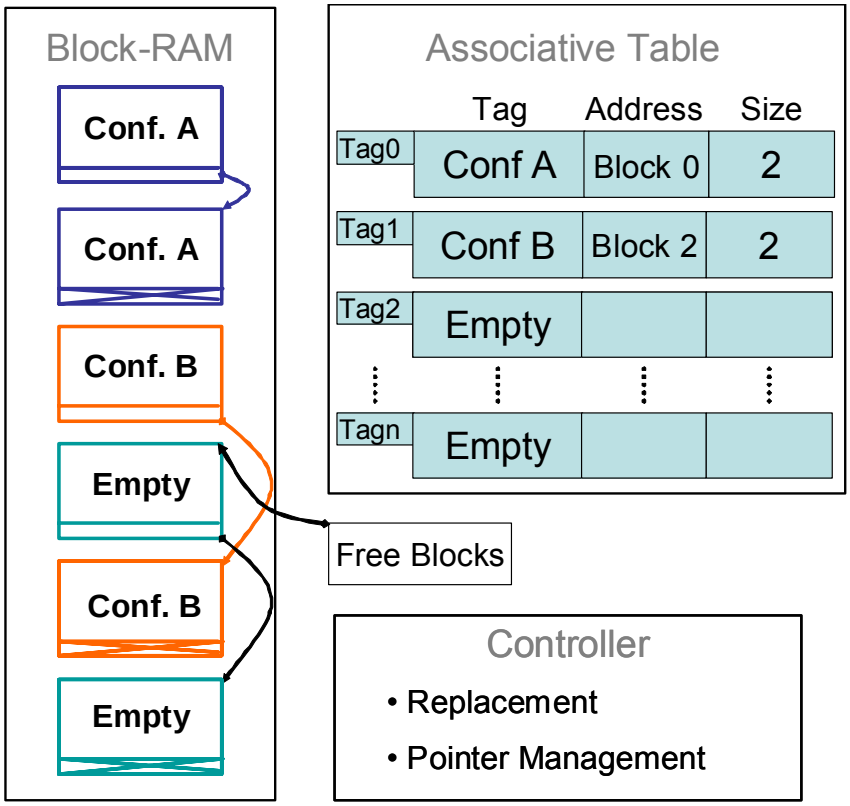


Figure 101. On-chip configuration memory.

In order to store a configuration the controller will use as many memory blocks as needed. These blocks will be linked using pointers. In addition, an associative table is used to identify whether or not a configuration can be fetched from the on-chip module. If the configuration is found, the table provides the initial address.

Appended A:
Resumen en Español

A.1. Introducción

Las aplicaciones multimedia actuales, como las aplicaciones de vídeo digital y los juegos en tiempo real con gráficos en tres dimensiones, se caracterizan por un comportamiento muy dinámico, con una carga de trabajo que puede variar varios órdenes de magnitud en tiempo de ejecución. Además, típicamente estas variaciones dependen de la interacción de las aplicaciones con los usuarios, de los datos de entrada, o incluso de las interacciones con otras aplicaciones, por lo que frecuentemente no resultan predecibles en tiempo de diseño.

Este tipo de aplicaciones comenzaron desarrollándose únicamente para computadores de sobremesa o consolas especialmente optimizadas para ello con

una gran capacidad de cálculo. Pero en los últimos años han comenzado a incluirse dentro de sistemas empotrados (como por ejemplo los teléfonos móviles y las PDAs) en los que los recursos disponibles están muy limitados por el reducido tamaño y precio de estos sistemas. Por si fuese poco, frecuentemente existe una segunda limitación impuesta por la dependencia de una batería con una capacidad muy reducida.

La migración de las aplicaciones multimedia a los sistemas empotrados comenzó con aplicaciones muy sencillas, pero cada vez se demanda la inclusión de aplicaciones más y más complejas. Para lidiar con estas aplicaciones es necesaria una plataforma con gran potencia computacional, pero también muy flexible que sea capaz de adaptarse eficientemente a los continuos cambios en la carga de trabajo. Conseguir la potencia de cálculo necesaria con un número de recursos limitado por las restricciones de tamaño y sin incurrir en un consumo de energía excesivo es el reto de los diseñadores que trabajan con sistemas empotrados.

Para implementar un algoritmo concreto la mejor solución tanto en rendimiento como en consumo de energía es incluir en el sistema un circuito hardware específico (ASICs) diseñado para ejecutarlo de forma óptima. Sin embargo, las limitaciones de área y el continuo aumento del número de aplicaciones que deben incluirse en este tipo de sistemas provocan que, frecuentemente, no sea posible utilizar ASICs para conseguir la capacidad de computación necesaria para todos los algoritmos utilizados por estas aplicaciones. Por ejemplo, los teléfonos móviles han comenzado a incluir codificadores y decodificadores de vídeo, imagen y audio, aplicaciones con gráficos en tres dimensiones y soporte para diversos estándares de conexión inalámbrica. En un ordenador personal estas aplicaciones se optimizan incluyendo tarjetas de sonido, de vídeo, aceleradores gráficos y tarjetas para

conexión inalámbrica, que incluyen ASICs para proporcionar el rendimiento necesario. Sin embargo, en los sistemas empotrados no hay suficiente espacio para incluir estos elementos.

Además, cada vez es más frecuente que se permita a los usuarios aumentar la funcionalidad del sistema, incluyendo nuevas aplicaciones que no eran conocidas en tiempo de diseño. Sin embargo los sistemas empotrados no suelen permitir que se les añada hardware a posteriori (en general porque no hay ningún espacio libre para ello). Por tanto resulta imposible incluir ASICs para optimizar la ejecución de dichas aplicaciones.

La opción más común para proporcionar flexibilidad a un sistema empotrado consiste en incluir un procesador. De hecho existen múltiples arquitecturas especialmente diseñadas para sistemas empotrados. Estos procesadores tienen un consumo de energía reducido, pero a costa de perder capacidad computacional con respecto a un procesador superescalar avanzado por lo que difícilmente pueden alcanzar el rendimiento que las aplicaciones actuales necesitan. Por tanto, resulta necesaria la inclusión de aceleradores hardware, pero, en general, no es factible satisfacer totalmente esta necesidad incluyendo ASICs.

El hardware dinámicamente reconfigurable (DRHW) tiene las características ideales para resolver este problema ya que, por un lado, puede alcanzar el rendimiento necesario, al permitir implementar circuitos que aprovechen al máximo el paralelismo de cada tarea de la aplicación; y por otro, un mismo recurso de hardware reconfigurable puede utilizarse como acelerador para un número de aplicaciones virtualmente ilimitado (en la práctica el número de aplicaciones que soporte

únicamente está limitado por el espacio de memoria asignado a almacenar las configuraciones).

Utilizando la posibilidad de reconfiguración parcial, la funcionalidad de los recursos hardware puede variarse en tiempo de ejecución para adaptarse a los requerimientos variables de las aplicaciones multimedia. Además, las plataformas reconfigurables actuales (principalmente FPGAs) han incluido recientemente características muy interesantes, como soporte para incluir bloques prediseñados (IPs), o recursos HW como multiplicadores y bloques de memoria distribuidos, que permiten aumentar su rendimiento y simplificar el proceso de diseño.

Además, algunas de las plataformas reconfigurables más recientes incluyen dentro de un único chip, no sólo recursos HW reconfigurables, sino también procesadores. Este tipo de plataformas heterogéneas permite utilizar únicamente los procesadores mientras los requisitos de rendimiento sean bajos y utilizar los recursos reconfigurables para cumplir los requisitos de las aplicaciones más exigentes. De esta forma, utilizando una política de ahorro de energía que mantenga los elementos que no se estén utilizando en un estado de bajo consumo, se puede conseguir una plataforma con un consumo de energía medio comparable a los procesadores para sistemas empotrados, y que a la vez pueda proporcionar el rendimiento pico que demandan las aplicaciones.

Sin embargo, actualmente los fabricantes de plataformas reconfigurables no proporcionan un entorno de trabajo en el que se puedan aprovechar fácilmente las ventajas del hardware reconfigurable. En concreto, no existe ningún soporte comercial específico para hardware reconfigurable que permita gestionar de forma sencilla la creación de tareas de forma dinámica, ni la sincronización y comunicación de estas tareas entre sí y con el resto de elementos del sistema.

Cuando se utilizan procesadores convencionales, un sistema operativo del tipo RTOS (Real Time Operating Systems), especialmente diseñados para sistemas empotrados, suele ser el encargado de proporcionar este soporte. Sin embargo ningún RTOS comercial es capaz de gestionar los recursos de HW reconfigurable, por lo que el diseñador debe crear un módulo que realice esta gestión.

Esta falta de soporte ha provocado que prácticamente ninguna plataforma comercial actual utilice la reconfiguración parcial en tiempo de ejecución, aunque los recursos dinámicamente reconfigurables comerciales soporten este tipo de reconfiguración desde finales de la década de los noventa.

En [MBVL02] se presenta un modelo de hardware dinámicamente reconfigurable basado en una red de interconexión dentro del chip que proporciona un soporte específico para gestionar la creación dinámica de tareas en los recursos de hardware reconfigurable. Este modelo se denomina ICN (Interconnection Network) e incluye soporte hardware para un sistema operativo (OS4RS, Operating Systems for Reconfigurable Systems, [MNCV03]), de forma que no sólo permite crear y asignar tareas a los recursos reconfigurables en tiempo de ejecución, sino que también proporciona soporte para garantizar que las tareas puedan comunicarse y sincronizarse entre sí, y que el sistema operativo pueda controlar su ejecución.

Una de las características principales del modelo ICN es que proporciona el soporte necesario para que se puedan gestionar los recursos HW y SW de una forma uniforme, convirtiendo los recursos reconfigurables en un conjunto de elementos de procesamiento (llamados unidades reconfigurables) a los que el sistema operativo puede asignar tareas de la misma forma que las asigna a los

procesadores (para utilizar el modelo ICN al menos un procesador debe estar siempre presente en el sistema, dado que debe ejecutar el sistema operativo).

Una vez adoptado el modelo ICN para proporcionar el soporte para la gestión de las tareas en HW reconfigurable, queda todavía pendiente para el diseñador desarrollar un módulo que decida dónde y cuándo se debe ejecutar cada tarea. Diseñar un planificador de tareas representa un trabajo demasiado complejo para llevarlo a cabo comenzando desde cero, especialmente si debe tratar con aplicaciones dinámicas cuyo comportamiento resulta impredecible en tiempo de diseño. En su lugar, dado que el modelo ICN trata a los recursos HW reconfigurables como si fuesen una plataforma multiprocesador, es posible adaptar un planificador de tareas inicialmente diseñado para plataformas con múltiples procesadores para que también funcione con recursos HW reconfigurables.

Dado que en los sistemas empotrados el consumo de energía es uno de los mayores problemas, el planificador de tareas debe ser capaz de gestionar los distintos recursos de forma que se proporcione la potencia de cálculo necesaria en cada instante, pero con el menor consumo posible de energía. Para realizar esta función hemos seleccionado el entorno de planificación TCM (Task Concurrency Management, [YWMC01]), que aporta flexibilidad de asignación y planificación de las tareas en tiempo de ejecución y a la vez genera una penalización mínima al tomar sus decisiones basándose en información precalculada en tiempo de diseño. Además TCM se diseñó desde el principio para sistemas heterogéneos (si bien hasta ahora únicamente se utilizaba para multiprocesadores), por lo que resulta relativamente sencillo incluir nuevos tipos de recursos.

Trabajando juntos, el modelo ICN y el entorno TCM aportan el soporte ideal para utilizar los recursos reconfigurables de forma eficiente. Haciendo posible que se puedan asignar tareas en tiempo de ejecución a los recursos reconfigurables, de la misma manera que se asignan a un procesador, gracias a la reconfiguración parcial. De esta forma se puede aprovechar la flexibilidad que proporcionan los recursos de hardware reconfigurable para conseguir una plataforma capaz de adaptarse en tiempo de ejecución a las exigentes demandas de las aplicaciones actuales.

Sin embargo utilizar la reconfiguración parcial genera una penalización considerable tanto en tiempo como en consumo de energía. Esta penalización no resulta aceptable para muchas de las aplicaciones multimedia actuales que, debido a su comportamiento dinámico, pueden demandar reconfiguraciones cada pocos milisegundos. Además, típicamente los planificadores de tareas no tienen en cuenta esta penalización, dado que en entornos multiprocesadores se puede comenzar a ejecutar una nueva tarea en tan solo algunos microsegundos siempre que esta esté almacenada en la memoria local del procesador. Por tanto, para ampliar un planificador de tareas inicialmente diseñado para procesadores (como en este caso TCM), para que gestione también recursos de hardware reconfigurable resulta imprescindible incluir un soporte específico para minimizar la penalización que dicha reconfiguración genera tanto en tiempo como en consumo de energía. Desarrollar este soporte es el objetivo que persigue el trabajo presentado en esta tesis.

A.1.1. Modelo de la arquitectura objetivo

La figura siguiente presenta la arquitectura objetivo con la que se ha trabajado durante el desarrollo de esta tesis. Esta plataforma está compuesta por un número variable de procesadores, algunos de los cuales son en realidad recursos de HW reconfigurable (en la figura se les llama unidades reconfigurables, UR). Desde el punto de vista del planificador de tareas las URs no se diferencian de cualquier otro elemento de procesamiento. Así el planificador recibe un conjunto de tareas a ejecutar, cada una de ellas representada como un grafo de tareas que incluye algunas restricciones temporales. A partir de estos grafos el planificador reparte las tareas entre lo procesadores tratando de cumplir las restricciones de tiempo y a la vez de elegir la asignación de tareas que consuma menos energía. Para ello junto a cada tarea se incluyen estimaciones sobre su tiempo de ejecución y su consumo de energía en cada uno de los procesadores presentes en el sistema.

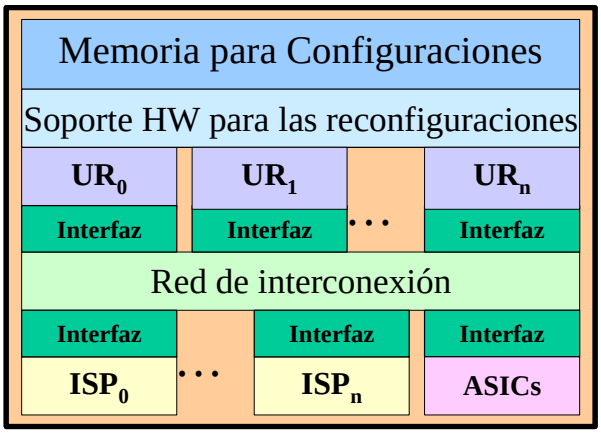


Figura 1. Arquitectura objetivo.

Actualmente ya existen arquitecturas comerciales (como la familia VIRTEX2-PRO de XILINX [Xili05] o la familia FPSLIC de Atmel [Atme05]) que encajan con este esquema de multiprocesadores heterogéneos en un único chip. Además debido al

crecimiento exponencial del número de transistores por chip previsto para los próximos años, parece evidente que cada vez va a ser posible incluir arquitecturas más y más complejas en un chip. De hecho, el diseño de sistemas completos dentro de un único chip (normalmente llamados SoC: System-on-a-Chip) es actualmente una de las líneas de investigación más activas en el área de los sistemas empotrados. La heterogeneidad de esta arquitectura no tiene porqué limitarse a dos tipos de procesadores: ISPs (Instruction Set Processors) y URs. Dentro de los ISPs puede existir una amplia variedad de elementos, incluyendo procesadores con arquitectura RISC, DSP, VLIW cada uno de los cuales puede adaptarse mejor a un tipo específico de tarea. Así mismo también puede existir heterogeneidad dentro de las unidades reconfigurables. En esta área el HW reconfigurable de grano fino (FPGAs) lidera ampliamente el mercado, sin embargo existen muchas otras arquitecturas emergentes, denominadas de grano grueso, que pueden resultar especialmente interesantes para los sistemas empotrados. Su consigna es proporcionar sólo “la flexibilidad necesaria” dado que toda la flexibilidad de programación presente en el HW reconfigurable provoca que no se alcancen los niveles de rendimiento propios de un ASIC y se consuma más energía. En este tipo de arquitecturas la flexibilidad se reduce normalmente obligando a operar a nivel de palabra y limitando el número de conexiones permitidas, mientras que en el HW reconfigurable de grano fino se puede trabajar a nivel de bit y se proporciona una conectividad muy alta. A cambio de estas limitaciones, los recursos de grano grueso prometen un mejor rendimiento y un menor consumo de energía para el rango de aplicaciones para el que están diseñados.

Se puede objetar que una plataforma con múltiples procesadores diseñados para sistemas empotrados resulta mucho menos potente, en términos de rendimiento, que un único procesador superescalar avanzado. Sin embargo, el rendimiento en este tipo de arquitecturas se obtiene a cambio de un consumo de energía insostenible para una plataforma que dependa de una batería reducida. Además la temperatura que alcanzan estos procesadores es tal, que necesitan voluminosos empaquetamientos y una serie de ventiladores para eliminar el exceso de calor que se genera.

Por otro lado las arquitecturas heterogéneas con múltiples procesadores presentan varias ventajas para su integración en sistemas empotrados:

- La primera, y más importante, es la flexibilidad que aportan. Como están formadas por elementos programables o configurables pueden utilizarse para un amplio rango de aplicaciones distintas. En el ámbito de los sistemas empotrados cada vez resulta más complejo y costoso desarrollar diseños específicos para cada producto. El coste derivado del desarrollo de un nuevo chip y especialmente el complejo proceso de verificación que requiere (actualmente el proceso de verificación consume más del 50% de todo el proceso de diseño) provoca que para aquellos productos cuya producción no sea muy elevada no resulte rentable diseñar una plataforma específica. En estos casos reutilizar una plataforma programable permite acelerar el proceso de diseño y reducir los costes. Además, incluso cuando se opta por desarrollar una

plataforma específica para un determinado producto, el reducido tiempo disponible para el proceso de diseño impide que se comience desde cero. En su lugar se elige entre una serie de componentes (IPs) disponibles por lo que, incluso en este caso, una plataforma multiprocesador a medida podría ser la solución idónea.

- La segunda ventaja consiste en proporcionar no una, sino múltiples posibilidades de asignación para una misma tarea (cada una a un procesador distinto). De esta manera el planificador de tareas puede elegir entre la asignación en tiempo de ejecución atendiendo a la situación en la que se encuentre el sistema. Por ejemplo, si una tarea se está ejecutando en solitario con unas restricciones temporales laxas, el planificador puede asignar sus tareas al procesador en el que consuman menos energía. Si más adelante aparecen nuevas tareas, y las restricciones se tornan más exigentes, el planificador distribuirá las tareas tratando de cumplir con estas restricciones a costa de aumentar el consumo de energía. De esta forma disponer de varias versiones de cada tarea con distintas características de consumo de energía y rendimiento permiten que el planificador de tareas pueda adaptar el consumo de energía a las necesidades del sistema.

- La heterogeneidad del sistema permite también generar plataformas que puedan ejecutar un amplio rango de algoritmos de forma eficiente. Así distintos tipos de tareas pueden encontrar un elemento de procesamiento adecuado para cada una de ellas. Por ejemplo, una tarea

dominada por estructuras de control puede ejecutarse en un procesador tradicional o en un VLIW, mientras que una tarea en la que se repitan una serie de cálculos intensivos sobre un amplio conjunto de datos puede ejecutarse en un DSP o en una unidad reconfigurable para aprovechar el alto nivel de paralelismo intrínseco.

- Debido al aumento del número de transistores por chip y al incremento consiguiente del número de elementos que se pueden integrar, el problema de la gestión de temperatura dentro del chip se ha hecho cada vez más y más importante. Esta temperatura no es constante, ya que dependiendo de la actividad de cada uno de los componentes se disipa distinta cantidad de energía. Cuando una región del chip experimenta un aumento de temperatura por encima de un determinado umbral, comienzan a aparecer efectos no deseados que provocan fallos en el funcionamiento del chip e incluso pueden llegar a dañarlo. En un sistema multiprocesador este problema se puede prevenir repartiendo equitativamente la carga de trabajo entre los distintos procesadores que pueden encontrarse en regiones distintas del chip y también desactivando aquellos procesadores que hayan alcanzado una temperatura peligrosamente cercana al umbral de seguridad.
- Otro de los efectos colaterales a la integración de cada vez más y más transistores en un único chip es el importante aumento de la posibilidad de errores en el funcionamiento del chip. Debido a la cercanía entre transistores, cualquier mínima perturbación o defecto de fabricación,

pueda provocar interferencias en una región del chip lo que a su vez puede provocan el malfuncionamiento de todo el conjunto. Conseguir eliminar el 100% de estos fallos es una tarea altamente complicada, especialmente cuando cada vez se exige reducir más y más el tiempo de diseño para poder adaptarse a la creciente competitividad del mercado. Un sistema con múltiples procesadores presenta una relativa tolerancia a este tipo de errores, ya que si se detecta el malfuncionamiento de uno de los procesadores este puede desactivarse y el resto del sistema puede continuar funcionando adecuadamente. Por supuesto para diseñar una plataforma realmente resistente a fallos debería incluirse a su vez redundancia en la red de interconexión y la jerarquía de memoria dado que los errores pueden aparecer en cualquiera de los componentes. Es interesante resaltar que debido a la regularidad y la redundancia inherente a los recursos reconfigurables, estos recursos son especialmente indicados para diseñar componentes con una alta tolerancia al malfuncionamiento de alguno de sus elementos.

Al igual que existen muchas e importantes ventajas por la utilización de plataformas heterogéneas multiprocesador, existen también inconvenientes considerables que dificultan la utilización de este tipo de arquitecturas:

- En primer lugar, aunque a priori las plataformas multiprocesador son fácilmente escalables, dado que se puede aumentar su potencia de cálculo sencillamente añadiendo más procesadores, en la práctica

resulta muy difícil extraer el suficiente paralelismo dentro de una aplicación para aprovechar toda esta capacidad de cálculo. Por tanto la potencia de cálculo utilizada es muy inferior a la potencia pico de la plataforma.

- El rendimiento de un multiprocesador puede degradarse ampliamente por la creación de cuellos de botella debidos a la saturación de la red de interconexión y a los accesos a elementos de memoria compartidos.
- Desarrollar varias versiones de una misma tarea conlleva un esfuerzo extra durante el proceso de diseño. Especialmente si cada versión se debe optimizar manualmente.
- Al existir diversas opciones de asignación para cada tarea, el planificador de tareas debe explorar un espacio de diseño muy amplio con muchas posibilidades que considerar y por tanto, el tiempo de cálculo podría ser muy elevado. Además, para trabajar con aplicaciones con comportamiento dinámico, la planificación de tareas debe realizarse al menos parcialmente en tiempo de ejecución, con lo que el tiempo necesario para calcular la planificación podría generar una penalización significativa en el rendimiento del sistema.

El primero de los problemas, la extracción del paralelismo de la aplicación, resulta mucho más sencillo cuando se utilizan los nuevos estándares para

aplicaciones multimedia en los que se trabaja con un conjunto de objetos, cada uno de los cuales se procesa de forma casi independiente del resto.

Para el segundo problema existen herramientas que, tras imponer una serie de restricciones a las comunicaciones y a los accesos a elementos compartidos en tiempo de diseño, son capaces de prevenir la creación de los cuellos de botella.

Para la creación de distintas versiones de una misma tarea para cada uno de los procesadores existen actualmente algunos entornos de trabajo capaces de generarlas automáticamente. OCAPI-XL es capaz de generar a partir de una especificación común dos versiones funcionalmente equivalentes de una misma tarea. La primera versión es código C que puede compilarse para cualquier procesador. La segunda versión es VHDL sintetizable que puede convertirse en un mapa de bits para una FPGA determinada.

Por último, para realizar la planificación de tareas sin generar una penalización excesiva en tiempo de ejecución se puede trabajar con planificadores híbridos. Estos planificadores constan de una componente en tiempo de diseño y otra en tiempo de ejecución de forma que casi todos los cálculos se realizan en tiempo de diseño generando distintas planificaciones. En tiempo de ejecución, el planificador sólo puede elegir entre una de las planificaciones precalculadas, con lo que el espacio de exploración está muy limitado y el tiempo de cálculo necesario para tomar las decisiones resulta aceptable.

A.1.2. Entorno de trabajo

Como ya he comentado anteriormente, el hardware dinámicamente reconfigurable (como por ejemplo una FPGA) puede aportar la potencia y la flexibilidad que demandan las plataformas multimedia actuales, especialmente en los sistemas empotrados donde el área está muy limitada y por tanto no se pueden incluir aceleradores hardware especializados para cada posible aplicación. Además, los recursos reconfigurables son cada vez más potentes y más baratos, con lo que su uso está creciendo de forma considerable.

Sin embargo, hay tres razones que están evitando que el uso de estos recursos sea todavía más amplio. Estas razones son:

- La falta de soporte para utilizar la reconfiguración parcial en tiempo de ejecución. Si el diseñador quiere utilizar la reconfiguración parcial para cargar una tarea en tiempo de ejecución en una porción de una FPGA, primero debe resolver cómo interconectar físicamente esta tarea para que pueda acceder a los recursos necesarios sin alterar las conexiones de las que ya se encontraban cargadas previamente y también cómo garantizar la comunicación entre esta tarea y el resto de los recursos (es decir definir un protocolo de comunicación que sea capaz de adaptarse a la aparición de esta tarea). Este problema es lo suficientemente complejo para que, a pesar de que las FPGAs permiten la reconfiguración parcial desde finales de la década de los noventa, todavía no exista ninguna plataforma comercial fácilmente utilizable.

- El hardware reconfigurable es menos eficiente energéticamente que los ASICs. Este consumo extra de energía es el precio que se paga por la flexibilidad de la plataforma. Típicamente se menciona que la diferencia llega a ser de un orden de magnitud. Por tanto, para que la inclusión de elementos reconfigurables sea aceptable desde un punto de vista energético, estos deben de gestionarse teniendo en cuenta cómo minimizar su consumo de energía.
- Para muchas aplicaciones el tiempo de reconfiguración es demasiado grande para que merezca la pena utilizar la reconfiguración en tiempo de ejecución. Por ejemplo, para reconfigurar una décima parte de una FPGA Virtex2 v6000 se necesitan 4 ms si se utiliza una frecuencia de reloj de 50 MHz (la frecuencia máxima teórica que soporta esta FPGA es 66 MHz, sin embargo no es recomendable alcanzarla [Xili05]).

Existen actualmente en el mercado dos soluciones independientes para los dos primeros problemas. En primer lugar el modelo ICN (InterConnection Network) de FPGA, basado en una red de interconexión, proporciona el soporte necesario para llevar a cabo reconfiguraciones parciales en tiempo de ejecución de una forma sencilla. Por otro lado el entorno de planificación TCM (Task Concurrency Management) permite planificar las tareas en entornos multiprocesadores heterogéneos de forma que se minimice el consumo energético a la vez que se cumplen las restricciones temporales.

En esta tesis se utiliza el entorno de planificación de tareas TCM y el modelo ICN para HW reconfigurable colaborando conjuntamente para proporcionar el soporte necesario para la gestión dinámica y planificación de tareas sobre HW. A raíz de proponer y experimentar con este entorno de trabajo se identificó que el tercer problema, el impacto que las reconfiguraciones parciales demandadas por aplicaciones con un comportamiento muy dinámico, podía por sí solo degradar la ejecución y el consumo de energía del sistema de forma considerable, hasta el punto de que no resulta beneficioso en muchos casos asignar tareas a las unidades reconfigurables.

A.1.2.1. El modelo ICN para hardware dinámicamente reconfigurable

El modelo ICN desarrollado por el IMEC ([MBVL02], [MMBM03], [BMNM03]) proporciona el soporte necesario para llevar a cabo reconfiguraciones parciales en tiempo de ejecución.

El modelo ICN divide la FPGA en un conjunto de unidades reconfigurables idénticas, en cada una de las cuales puede ejecutarse una tarea. Cada una de estas unidades incluye un interfaz de comunicación fijo. Este interfaz permite la comunicación entre unidades usando primitivas de paso de mensajes sobre una red de interconexión que se encuentra integrada dentro de la FPGA. En la versión actual la red de interconexión así como los interfaces se implementan utilizando los recursos de interconexión y almacenamiento de la FPGA. Sin embargo, dado que estos elementos no se reconfiguran, en una versión futura podría implementarse con

una red fija no reconfigurable para reducir el consumo de energía y mejorar el rendimiento, especialmente teniendo en cuenta que los interfaces y la red implementan una funcionalidad crítica para el rendimiento del sistema.

La aportación más importante del modelo ICN es que permite utilizar de forma efectiva la capacidad de reconfiguración parcial para cargar tareas en una FPGA en tiempo de ejecución.

Aunque las FPGAs actuales permiten la carga de nuevas tareas en tiempo de ejecución utilizando la capacidad de reconfiguración parcial, el proceso de diseño clásico no es compatible con esta posibilidad. Para utilizar las posibilidades que ofrece la reconfiguración parcial se tiene que conseguir que las tareas puedan cargarse en tiempo de ejecución para ajustarse a las demandas del sistema. La posición dónde se va a cargar la tarea debe decidirse en tiempo de ejecución para poder aprovechar los recursos de la FPGA que se encuentren inactivos en un determinado instante. Sin embargo, en principio, esto implica que tanto el proceso de asignación como el proceso de interconexionado deben realizarse también en tiempo de ejecución, una vez que se conoce dónde se va a cargar la tarea y qué elementos están siendo usados por otras tareas. Lamentablemente, el proceso de asignación e interconexionado es extremadamente complejo y puede llevar desde segundos hasta horas de cálculo, con lo que utilizar la reconfiguración parcial prácticamente dejaría de tener ningún sentido.

El modelo ICN soluciona este problema dividiendo la FPGA en unidades idénticas con un interfaz de comunicaciones fijo y conocido en tiempo de diseño. De esta forma el diseñador puede ejecutar en tiempo de diseño una herramienta de

asignación e interconexionado, imponiendo que la tarea se implemente en una de estas unidades. Dado que todas las unidades son idénticas y sus características han sido tenidas en cuenta en tiempo de diseño, más adelante, en tiempo de ejecución, la tarea podrá cargarse en cualquiera de ellas sin que sea necesario realizar ningún cálculo.

Otra aportación fundamental de ICN es que define un protocolo de comunicación entre tareas y proporciona un entorno que lo soporta: la red de interconexión. Este protocolo es válido tanto para tareas ejecutándose en las unidades reconfigurables como para las que se ejecuten en SW ya que ambas utilizan las mismas directivas de paso de mensajes.

Además, el interfaz de comunicación provee el soporte para ejecutar las primitivas básicas de paso de mensajes.

El interfaz de comunicación, permite la sincronización y comunicación entre tareas de una forma clara y sencilla para el diseñador, que puede limitarse a utilizar las primitivas soportadas por el interfaz en lugar de tener que diseñar y verificar sus propios mecanismos de comunicación.

A.1.2.2 Entorno de planificación de tareas en sistemas con multiprocesadores

TCM, [YDCV00], [YWMC01], [WMYP01]], [MJSY02], [LWMV02], [YaCa03], es un entorno de gestión de concurrencia y planificación de tareas para plataformas con múltiples procesadores que se está desarrollando en el IMEC desde 1999.

TCM se basa en tres ideas principales: modelar la aplicación de forma jerárquica en dos niveles, utilizar varios escenarios para modelar una misma tarea y realizar la planificación de las tareas en dos pasos, uno en tiempo de diseño y otro en tiempo de ejecución.

Las aplicaciones se modelan de forma jerárquica en dos niveles:

En el nivel superior la aplicación se describe como un conjunto de tareas que interaccionan de forma dinámica, e incluso no determinista, entre sí. Para este nivel se utiliza el modelo MTG* [ThCa00] que es una extensión de las redes de Petri [Mura89] diseñada especialmente para representar este tipo de interacciones.

En el nivel inferior se describe cada una de las tareas como un grafo de tareas, donde cada tarea es un fragmento de código con entidad suficiente para que pueda ser asignada por separado a un procesador distinto. En este nivel no se permite ningún evento no determinista, y el comportamiento dinámico se encuentra limitado a bucles y condiciones que únicamente dependan de los datos de entrada.

Resumiendo, tan sólo se permite un comportamiento dinámico limitado dentro de cada grafo de tareas. Por tanto, la mayor parte del comportamiento dinámico y sobre todo el comportamiento imprevisible debe quedar fuera, en el nivel superior. El objetivo de esta partición es separar la parte de la especificación de la aplicación que puede ser tratada y optimizada en tiempo de diseño, de la que sólo puede tratarse en tiempo de ejecución.

Dado que se permite cierto dinamismo dentro de cada grafo de tareas resulta imposible caracterizar a cada tarea del grafo con un tiempo de ejecución único. Por ejemplo, si existen condiciones dentro del grafo, el tiempo de ejecución variará según los datos de entrada. Para solucionar este problema las aproximaciones más comunes son caracterizar cada tarea con el máximo tiempo de ejecución posible (worst-case) o caracterizarla con el

tiempo de ejecución medio. La primera opción supone un escenario demasiado pesimista, generalmente de forma injustificada ya que el peor caso posible puede ser muy poco común. La segunda opción es más realista, pero puede llevar a incumplir las restricciones temporales cada vez que el tiempo de ejecución se aleje del tiempo de ejecución medio. En TCM la solución consiste en utilizar varios escenarios para cada tarea. Un escenario representa la ejecución de la tarea para un determinado rango de valores de entrada. Cada escenario se representa con su propio grafo de tareas, así por cada tarea habrá tantos grafos como escenarios se hayan definido. El número de escenarios se debe elegir de forma cuidadosa para que no resulte excesivo. Para limitar el número de escenarios se pueden agrupar los escenarios parecidos. Dentro de cada grupo se utiliza una aproximación worst-case.

El proceso de planificación en TCM comienza a partir de los grafos de tareas. En TCM cada tarea tiene tantos grafos asociados como escenarios se hayan definido. Como estos grafos tienen un comportamiento principalmente estático, resulta posible analizarlos en tiempo de diseño. Así, el planificador de tareas en tiempo de diseño explora para cada grafo las distintas posibilidades de asignación y planificación de las tareas. Sin embargo, dado que no conoce las condiciones en las que la tarea va a ejecutarse (estas condiciones sólo se conocen en tiempo de ejecución), no puede predecir cuál será la solución más adecuada en cada instante. Por ello, en lugar de calcular una solución única, este planificador genera un conjunto de soluciones con distintas características de rendimiento y consumo de energía. Cada una de estas soluciones es una asignación de las tareas a los distintos procesadores y una planificación de su ejecución. A este conjunto se le llama curva de Pareto, y a cada solución punto de Pareto. Cada punto representa una solución pseudo-óptima, de forma que para cada uno de ellos se cumple que ninguna de las demás soluciones exploradas es mejor en los dos parámetros de optimización considerados:

tiempo de ejecución y consumo de energía. De esta forma no se puede decir que un punto de Pareto sea mejor que otro, sino que cada uno se adapta mejor a una situación determinada. Por ejemplo, un punto de Pareto puede ser muy rápido, pero consumir mucha energía, mientras que otro es más lento pero tiene un consumo menor. Este tipo de curvas se utilizan frecuentemente para sistemas con diversos parámetros a optimizar [EsKO90].

Una vez que se han generado las curvas de Pareto para cada grafo, en tiempo de ejecución se seleccionan los grafos a ejecutar. Para ello se identifican las tareas activas y el escenario adecuado para cada una de ellas. A continuación se elige el punto de Pareto de cada grafo en función de las condiciones del sistema. Si la carga del sistema es muy grande, se elegirán puntos capaces de cumplir las restricciones temporales aunque consuman mucha energía, pero si más adelante, la carga de trabajo del sistema disminuye, los puntos seleccionados cambiarán para disminuir este consumo. En [YaCa03] y [YaCa04] se describen los algoritmos utilizados para la elección de los puntos de Pareto en tiempo de ejecución.

La idea básica de este proceso de planificación es proporcionar flexibilidad en tiempo de ejecución, pero al mismo tiempo ubicar en tiempo de diseño la exploración del espacio de soluciones. De esta forma se consigue flexibilidad sin incurrir en una penalización excesiva debida a la ejecución del planificador en tiempo de ejecución.

TCM se ha aplicado con éxito para planificar varias aplicaciones multimedia en plataformas con múltiples procesadores [WMYP01], [YaCa03], consiguiendo muy buenos resultados tanto a la hora de cumplir las restricciones temporales como a la hora de minimizar el consumo de energía.

A.1.3. Gestión de las reconfiguraciones en tiempo de ejecución

Un planificador diseñado para multiprocesadores (como los planificadores de TCM) no suele incluir el tiempo de carga de una tarea en el procesador, dado que si esta se encuentra en la memoria local este tiempo es poco significativo. Sin embargo, si uno o varios de los elementos de procesamiento son unidades reconfigurables el tiempo y la energía necesarios para cargar una tarea no sólo no es despreciable sino que en algunos casos puede ser incluso mayor que el tiempo de ejecución de la misma. Por tanto cualquier decisión que no tenga en cuenta este retardo puede llevar a resultados mucho peores de lo estimado en un principio.

Para paliar esta falta de precisión se han desarrollado una serie de módulos que actualizan la planificación elegida por el planificador de tareas incluyendo los retardos debidos a la carga de las tareas asignadas a las unidades reconfigurables.

Evidentemente, añadir estos retardos no es suficiente. Si no que también es necesario tomar decisiones (aunque siempre ciñéndose a la planificación elegida) que minimicen la influencia de estas cargas. En la figura 2 se puede ver el flujo de planificación de tareas en tiempo de ejecución utilizado en esta tesis.

Una vez que el planificador de tareas selecciona una planificación, el módulo de inicialización la analiza y genera las estructuras de datos que utilizan el resto de los módulos.

A continuación se trabaja con cada tarea de forma secuencial siguiendo el orden de la planificación inicial. Para cada tarea se aplican tres módulos: el de reutilización, el de precarga y finalmente el de reemplazo.

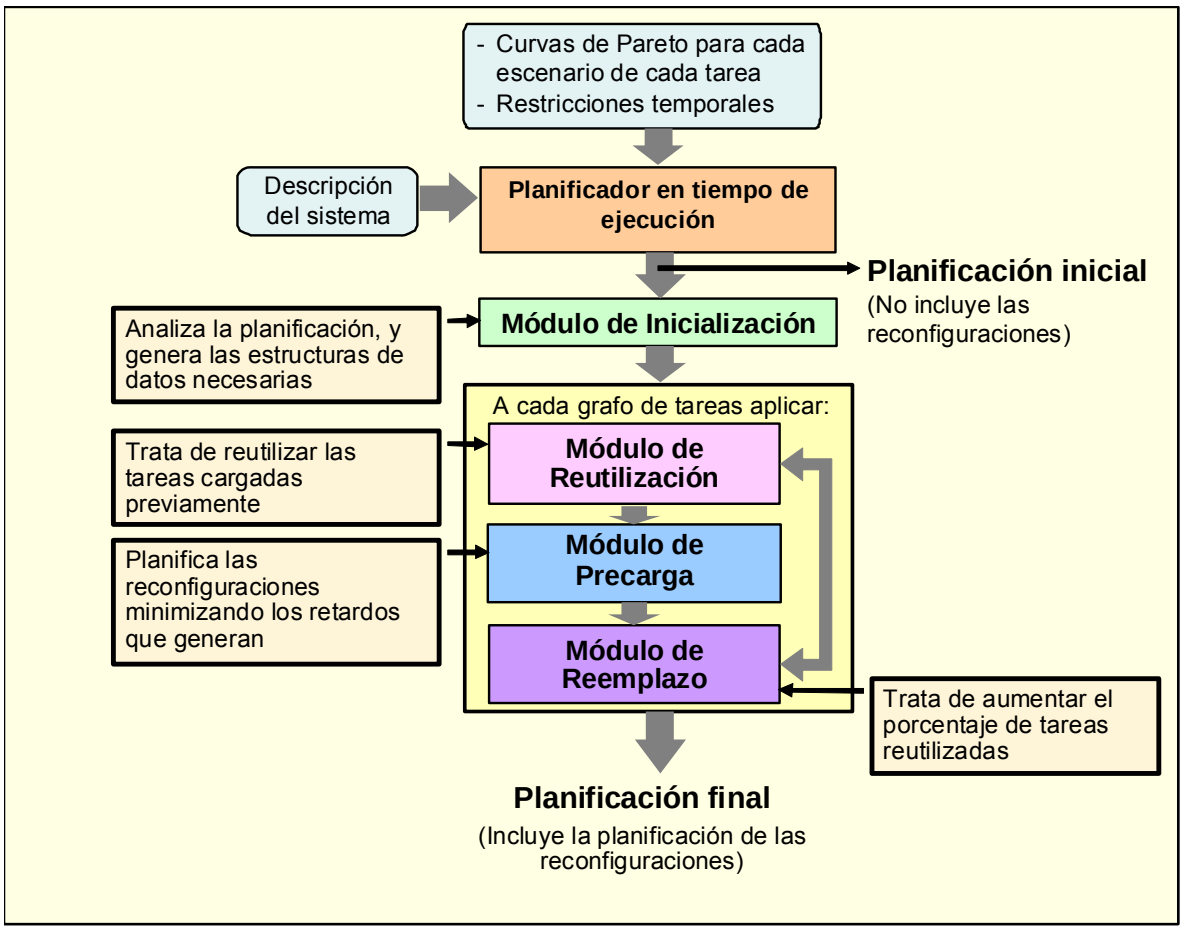


Figura 2. Flujo de planificación en tiempo de ejecución.

El módulo de reutilización comprueba si alguna de las tareas asignadas a las unidades de hardware reconfigurable se encuentra ya cargada. En ese caso la tarea puede ejecutarse directamente, sin necesidad de cargarla de nuevo.

El módulo de precarga planifica la carga de las tareas que no ha sido posible reutilizar tratando de realizar la carga de las tareas tan pronto como sea posible. Su objetivo es conseguir que las tareas se encuentren cargadas antes de que sea necesario ejecutarlas para que de esta forma el tiempo de carga no genere ninguna penalización.

Por último el módulo de reemplazo se ocupa de decidir en qué unidad deben cargarse las tareas. Su objetivo es aplicar una política de reemplazo que maximice el número de tareas reutilizadas. Así, cada vez que se debe cargar una tarea se trata de predecir cuál de las tareas cargadas tiene menos posibilidades de ser reutilizada en el futuro.

Estos tres módulos se aplican en tiempo de ejecución dado que si las aplicaciones tienen comportamiento dinámico resulta imposible predecir en tiempo de diseño qué tareas podrán reutilizarse. Y sin conocer qué tareas van a ser reutilizadas no se puede planificar de forma eficiente la carga de las tareas restantes. Aun así es posible utilizar en estos módulos algunos datos calculados en tiempo de diseño para reducir así su complejidad.

A.2. Objetivos

El objetivo fundamental de esta tesis consiste en desarrollar un conjunto de técnicas que permitan la gestión dinámica de las reconfiguraciones al mismo tiempo que se minimicen el máximo posible las desventajas de dicha reconfiguración.

El trabajo presentado en esta tesis pretende evitar que la latencia de la reconfiguración degrade el rendimiento del sistema, al mismo tiempo que intenta reducir al máximo posible la penalización en consumo de energía debida a las reconfiguraciones. Para ello hemos desarrollado un conjunto de técnicas de

planificación y propuesto un nuevo diseño para la jerarquía de memoria de configuraciones.

La jerarquía de la memoria de configuraciones para el HW reconfigurable este normalmente formada por un soporte reconfigurable que almacena las configuraciones que están lista para ser ejecutada y una memoria off-chip en la cual permanecen almacenadas todas las configuraciones de la aplicación asignadas para ser ejecutadas en el HW reconfigurable. Este es el esquema que se encuentra normalmente presente en las arquitecturas de grano fino, como las FPGAs. Un mejor interesante, que a menudo es utilizada en arquitecturas de grano grueso, consiste en añadir una pequeña memoria de configuraciones intermedia dentro del chip, en la cual se encuentran almacenadas las configuraciones de las tareas en ejecución. Esta memoria de configuraciones es crítica para todo el sistema, no solo por el elevado tráfico de configuraciones que registra en la ejecución de aplicaciones dinámicas, si no también por el consumo de energía que conlleva su utilización.

En este tesis proponemos incluir en la dentro de la jerarquía de la memoria de configuraciones una capa on-chip heterogénea, que remplazará la típica memoria off-chip de los sistemas de grano fino. Esta capa de memoria esta formada por dos módulos de memoria distintos, uno optimizado para alto rendimiento, y el otro optimizado para baja consumo de energía. Por lo tanto, la jerarquía de memoria de configuraciones que proponemos en esta tesis proporciona una rápida reconfiguración cuando es necesario, al mismo tiempo que la posibilidad de conseguir un importante ahorro de energía cuando esta velocidad de reconfiguración no es necesaria.

Añadir este nivel on-chip extra de memoria dentro de la jerarquía de memoria de configuraciones aumenta el espacio de búsqueda, ya que ahora es necesario decidir que configuraciones han de ser asignadas en cada uno de los diferentes módulos de esta jerarquía. Para poder optimizar las ventajas potencial que conlleva la inclusión de este nivel heterogéneo dentro de la jerarquía de memoria de configuraciones, en este tesis hemos desarrollado varios algoritmos da mapeado de configuraciones que han sido incluidos dentro del planificador. Nuestros algoritmos sistemáticos de mapeo analizan en tiempo de diseño las características de las diferentes tareas que conforman la aplicación que va a ser ejecutada, e interaccionan con el modulo de precarga. Los algoritmos de mapeo deben decidir si las configuraciones antes de ser cargadas en el HW reconfigurable para ser ejecutadas van almacenarse antes de su ejecución en el modulo de memoria optimizado en rendimiento o en el optimizado en consumo de energía del nivel on-chip de la jerarquía de memoria, o bien si es necesario van a ser siempre desde la memoria off-chip de configuraciones. Almacenar una configuración en el modulo de memoria on-chip optimizado en consumo de energía reduce la energía consumida por la reconfiguración pero puede producir un aumento del tiempo necesario para completar la ejecución de la aplicación. El objetivo de los diferentes algoritmos de mapeo propuestos en esta tesis es identificar la partición de configuraciones que minimiza el gasto de energía debido a la configuración sin incrementar el tiempo de ejecución de la aplicación. Para mapear o asignar adecuadamente cada una de las configuraciones a los diferentes módulos de memoria que componen la jerarquía de configuraciones es fundamental identificar cuales son las tareas que tienen un mayor

impacto en el rendimiento global del sistema, y asignarlas al modulo de memoria on-chip optimizado en rendimiento, mientras que las tareas restantes serán cargadas desde el modulo de memoria on-chip optimizado en consumo de energía, o desde memoria externa si fuese necesario.

A.3. Aportaciones Fundamentales

Como ya hemos comentado nuestro algoritmos sistemáticos de mapeo de configuraciones analizan en tiempo de diseño las características de los grafos que conforman las aplicaciones que van a ser ejecutadas interactuando con el modulo de precarga. La tarea de los algoritmos de mapeo consiste en decidir si las configuraciones han de ser almacenadas en la memoria optimizada en consumo de energía, en la optimizada en rendimiento, si es necesario en la memoria externa, para ser cargadas desde allí en los recursos configurables antes de ser ejecutadas. Almacenar una configuración en el modulo de memoria optimizado en consumo de energía supone una reducción en dicho consumo pero incrementa el coste en tiempo de ejecución. Mientras que asignar una configuración en memoria externa no solo implica un incremento en la latencia de reconfiguración, si no que también significa un incremento de la energía necesaria para llevar a cabo la reconfiguración. Sin embargo, en determinados casos es necesario cargar configuraciones desde memoria externa para evitar problemas de trasiego de tareas en el nivel on-chip de la jerarquía de memoria de configuraciones.

La meta de nuestros algoritmos de mapeo es identificar una partición de las configuraciones que minimice la penalización en consumo de energía debido a la reconfiguración mientras que obtienen el rendimiento deseado durante la ejecución del sistema. Por lo tanto, es fundamental que los algoritmos identifiquen cuales son las tareas que tienen mayor impacto en el rendimiento de todo el sistema, y que las asignen al modulo de memoria on-chip optimizado en rendimiento mientras que las tareas restantes son almacenadas en la memoria on-chip optimizada en consumo de energía, o en memoria externa teniendo en cuenta las características del sistema de ejecución.

En esta tesis hemos desarrollado tres algoritmos de mapeo diferentes. El primero de ellos trabaja bajo la suposición de que todas las configuraciones de las tareas que van a ser ejecutadas en el HW reconfigurable pueden ser precargadas todas juntas en el nivel de memoria on-chip antes de comenzar la ejecución de la aplicación. La idea básica de este algoritmo de mapeo es identificar una partición de tareas que minimice el consumo de energía debido a la reconfiguración sin que se produzca ninguna degradación en el rendimiento. Para llevarlo a cabo, el algoritmo de mapeo identifica aquellas tareas cuya latencia de carga no puede ser ocultada y las asigna al modulo de memoria on-chip optimizado en rendimiento para minimizar la penalización que provocan en el rendimiento del sistema. Las tareas restantes son asignadas al modulo de memoria optimizado en consumo de energía. Este mapeo representa el caso óptimo, tanto en rendimiento como en consumo de energía. Sin embargo, los recursos on-chip están a menudo limitados para los sistemas empotrados, y si el número y tamaño de las configuraciones excede la capacidad

del nivel on-chip de la jerarquía de memoria de configuraciones esta aproximación puede conducirnos a resultados muy alejados de la solución óptima en tiempo de ejecución debido al efecto de trasiego de configuraciones dentro de la jerarquía de memoria. El trasiego de configuraciones tiene lugar cuando las configuraciones son sistemáticamente traídas y remplazadas en el nivel de memoria on-chip. El problema del trasiego es un problema muy conocido y ampliamente estudiado en el campo de la gestión de memoria.

Para abordar este problema, hemos ampliado las el algoritmo anterior estableciendo restricciones a nivel de grafo de tareas para garantizar que las configuraciones que son asignadas a cada uno de los módulos de memoria on-chip no exceden la capacidad de almacenamiento de estos. El nuevo algoritmo de mapeo comparte con el objetivo principal del algoritmo anterior: encontrar una partición de tareas que proporcione el mejor rendimiento posible, al mismo tiempo que reduzca lo más posible el consumo de energía debido a la reconfiguración. La diferencia principal entre los dos algoritmos es que este último tiene en cuenta la capacidad de los módulos de memoria on-chip y el tamaño de las configuraciones. Si tenemos en cuenta que ambos módulos de memoria on-chip proporcionan normalmente un mejor rendimiento al mismo tiempo que consumen menos energía que la memoria externa, este algoritmo intenta siempre maximiza la utilización del nivel on-chip de la memoria de configuraciones, asignando a ellas el mayor número posible de configuraciones. Si no existe suficiente espacio para almacenar todas las configuraciones en el nivel on-chip de la jerarquía de memoria de configuraciones, el algoritmo de mapeo asignara alguna de ellas al modulo de memoria externa para evitar el problema de

trasiego de configuraciones. Esta extensión resuelve el problema de trasiego de configuraciones siempre que exista un único grafo para ser ejecutado. Sin embargo, la aplicación de este sistema puede conducirnos a problemas de trasiego de configuraciones cuando se aplica a sistemas dinámico que permiten la ejecución entrelazada de varios grafos.

Para resolver este problema, hemos desarrollado un tercer algoritmo de mapeo de configuraciones orientado a sistemas dinámicos en los que diferente grafos de tareas son ejecutados un determinado número de veces, y sin que sepamos en tiempo de diseño que grafos van a ser ejecutados de forma concurrente. Debido a que en los sistemas de ejecución dinámicos no conocemos las condiciones reales de ejecución no podemos buscar una solución óptima. Por lo tanto, en este caso el algoritmo de mapeo de configuraciones busca una solución cumpla una restricción de rendimiento dada que no se puede superar, siempre que no existan conflictos en tiempo de ejecución, al mismo tiempo que genera la minima presión posible en el nivel de memoria on-chip de la jerarquía de memoria de configuraciones. Adicionalmente, el algoritmo trate de reducir al máximo posible el consumo de energía debido a la reconfiguración. En este caso, el algoritmo de mapeo no solo optimiza la ejecución de un único grafo de tareas, si no que intenta reducir la máximo posible los conflictos en la capa de on-chip de memoria, ya que estos conflicto puede generar importantes penalizaciones tanto en rendimiento como en consumo de energía. La idea principal de este ultimo algoritmo de mapeo básicamente consiste en analizar cada grafo de tareas por separado en tiempo de diseño para identificar que configuraciones puede ser cargadas de desde memoria

externa sin introducir ninguna degradación en el rendimiento de la ejecución de todo el grafo, o aquellas que introducen una penalización en el redimiendo que puede ser asumida por el sistema. Una vez el algoritmo ha identificado cuales son estas configuraciones, las etiqueta como tareas “no cacheable” para reducir la presión en la capa on-chip de la jerarquía de memoria de configuraciones.

Para probar nuestras propuesta, las hemos testeado con dos conjuntos de grafos de tareas diferentes: uno para una plataforma de grano fino, y otro para una plataforma de grano grueso. En el caso de la plataforma de grano fino hemos utilizado datos de la seri de Fugas Virtex de Xilinx [Xili09] ya que actualmente son las dominantes en el mercado de las plataformas reconfigurables de grano fino. Para grano grueso hemos elegido el entorno de simulación para arquitectura de grano grueso denominado CRISP (Configurable and Reconfigurable Instruction Set Processor) [MMSY07]. CRISP es una simulador de entrono académico que puede utilizarse para simular procesadores reconfigurables a nivel de instrucción de grano grueso.

Los resultados de los experimentos demuestran que, aplicando el algoritmo de mapeo de configuraciones adecuado, la jerarquía de memoria de configuraciones hibrida de bajo consumo de energía y alto rendimiento obtiene el rendimiento optimo mientras que reduce de forma drástica el consumo de energía debido a la reconfiguración, en comparación con un sistema que almacena toda las configuraciones para ser cargadas en el HW reconfigurable únicamente en una memoria externa. Además, esta aproximación obtiene el mismo rendimiento que un sistema que solo utilizase módulos de memoria de alto rendimiento, mientras que en

media consigue asignar la mitad de las tareas al modulo de memoria on-chip de bajo consumo, obteniendo reducciones significativas en el consumo de energía.

Con respecto a los dos últimos algoritmo de mapeo presentados, es importante resaltar que ninguno de ellos es mejor que el otro, ya que representan diferentes equilibrios entre la presión ejercida en el nivel on-chip de la jerarquía de memoria de configuraciones y el consumo de energía debido a las reconfiguraciones. El primer algoritmo minimiza el consumo de energía asignado el mayor número posible de configuraciones a los módulos de memoria on-chip. Únicamente esta activo un grafo de tareas para su ejecución el resultado obtenido será la solución óptima. Sin embargo, si varios grafos de tareas están compitiendo por los recursos del sistema, el efecto del trasiego de configuraciones puede degradar de forma drástica el rendimiento del sistema y aumentar considerablemente le consumo de energía. En estos casos se deberá aplicar el último algoritmo presentado, ya que proporciona el mismo rendimiento, peso asigna el mayor número posible de configuraciones ala memoria externa. Esta aproximación reduce la presión ejercida sobre el nivel on-chip de la jerarquía de memoria de configuraciones con un coste que se traduce en un aumento de la energía consumida en la reconfiguración. Por tanto es el diseñador el que debe decidir cual es el algoritmo de mapeo adecuado para cada entorno de ejecución.

A.4. Conclusiones

Claramente, los beneficios que del trabajo presentado en esta tesis dependen de la granularidad de las tareas y del coste de la reconfiguración. Si en algún caso las penalizaciones de la reconfiguración son poco significativas, nuestros algoritmos de mapeo no serían necesarios. Sin embargo, como se puede ver en los resultados obtenidos con los experimentos realizados en muchos casos la penalizaciones tiene un gran impacto tanto en el rendimiento como en consumo de energía del sistema. En estos casos el trabajo presentado en esta tesis consigue un elevado ahorro en la energía debida a la reconfiguración al mismo tiempo que cumple con el rendimiento requerido.

Publications

In chronological order:

[PRMC07a] E. Pérez Ramo, J. Resano, D. Mozos, and F. Catthoor, "A memory hierarchy for high-performance and energy-aware reconfigurable systems", IET Computers & Digital Techniques, Vol. 1, No. 5, pp. 565-571, 2007.

[PRMC07b] E. Perez-Ramo, J. Resano, D. Mozos, F. Cathoor, "Reducing the Reconfiguration Overhead: A Survey of Techniques", International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'07), 2007

[PeRe06] E. Perez-Ramo, J. Resano, "A Dual Cache for Performance and Energy Aware Reconfigurable HW", International Conference on Field Programmable Logic and Applications (FPL'06), 2006.

[PRMC06] E. Pérez, J. Resano, D. Mozos, F. Cathoor, "A Configuration Memory Hierarchy for Fast Reconfiguration with Reduced Energy Consumption Overhead", IEEE 13th Reconfigurable Architectures Workshop (RAW'06), 2006

[PRMM03a] M.E. Pérez, J.J. Resano, D. Mozos, H. Mecha, J. Septién, "Función de Coste Dinámica para Particionamiento Hw/Sw Multiobjetivo", Seminario Anual de Automática, Electrónica Industrial e Instrumentación (SAAEI'03), 2003.

[PRMC03b] M.E. Pérez , J.J. Resano, D. Mozos, H. Mecha, J. Septién "A Multi-Objective Dynamic Cost Function for Codesign Hardware/Software Partitioning", Design of circuits and integrated systems conference (DCISC 2003), 2003.

[RPMM03a] J.J. Resano, M.E. Pérez, D. Mozos, H. Mecha, J. Septién. "Analyzing Communication Overheads during Hardware/Software Partitioning", Elsevier Microelectronics Journal, Vol. 34, No. 11, pp.1001-1007, 2003.

[RPMM03b] J.J. Resano, M.E. Pérez, D. Mozos, H. Mecha, J. Septién, "A hardware/software partitioning and scheduling approach for embedded systems with low-power and high performance requirements". Proceedings of the 13th International Workshop on Power and Timing Modelling, Optimization and Simulation (PATMOS'03), Springer Verlag Lecture Notes on Computer Science, Vol. 2799, pp. 580-589, 2003.

[RPMM02a] J.J. Resano, M.E. Pérez, D. Mozos, H. Mecha, J. Septién. "Analyzing Communication Overheads during Hardware/Software Partitioning", Proceedings of the First International Workshop on Embedded System Codesign (ESCODES'02), pp. 16-21, 2002.

[RPMM02b] J.J. Resano, M.E. Pérez, D. Mozos, H. Mecha, J. Septién, “Analyzing Communication Overheads during Hardware/Software Partitioning”, First Internacional Workshop on Embedded System Codesign (ESCODES’02), pp. 16-21, 2002.

[RPMM02c] J.J. Resano, M.E. Pérez, D. Mozos, H. Mecha, J. Septién, “Planificación de las comunicaciones en un entorno de codiseño hardware/software”, Seminario Anual de Automática, Electrónica Industrial e Instrumentación (SAAEI'02), pp. 293-296, 2002.

[RPMM02d] J.J. Resano, M.E. Pérez, D. Mozos, H. Mecha, J. Septién, “Communication Scheduling Integration during Hardware/Software Partitioning”, Design of circuits and integrated systems conference (DCISC 2002), pp. 261-266, 2002.

References

In alphabetical order:

[ACGL92] A. Abnous, C. Christensen, J. Gray, J. Lenell, A. Naylor, and N. Bagherzadeh, “Design and implementation of TinyRISC microprocessor”, Microprocessors and Microsystems, Vol. 16, No. 4, pp. 187-194, 1992.

[Arm05] www.arm.com/support/ARMulator.html

[Atme05] www.atmel.com

[BBCS01] B. Benini, D. Bruni, M. Chinosi, C. Silvano, R. Zaccaria, and R. Zafalon, "A power modelling and estimation framework for VLIW-based embedded

systems", Proc. Int. Workshop-Power and Timing Modelling, Optimization and Simulation, pp. 26-28, 2001.

[Bela66] L.A. Belady, "A study of replacement algorithms for virtual storage computers", IBM Systems Journal, Vol. 5, No. 2, pp. 78-101, 1966.

[BJKM03] B. Blodget, P. James-Roxby, E. Keller, S. McMillan, and P. Sundararajan, "A self-reconfiguring platform", Proc. Field-Programmable Logic and Applications, pp. 565-574, 2003.

[BJOD02] F. Barat, M. Jayapala, P. Op de Beeck , and G. Deconinck , “Software pipelining for coarse grained reconfigurable instruction set processors”, Design Automation Conference (ASP-DAC'02), pp. 338–344, 2002.

[BJVL03] F. Barat, M. Jayapala, T. Vander Aa, R. Lauwereins, G. Deconinck, and H. Corporaal, "Low Power Coarse-Grained Reconfigurable Instruction Set Processor", Field Programmable Logic and Application (FPL'03), pp. 230-239, 2003.

[BJVL03] F. Barat, M. Jayapala, T. Vander, R. Lauwereins, G. Deconinck, and H. Corporaal, "Low power coarse-grained reconfigurable instruction set processor", Proc. Field-Programmable Logic and Applications, pp. 230-239, 2003.

[BMNM03] T.A. Bartic, J-Y. Mignolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde, R. Lauwereins, "Highly Scalable Network on Chip for Reconfigurable Systems Systems", Proceedings of the International Conference on System-On-Chip (SoC'03), pp. 79-82, 2003.

[Breb96] "G. Brebner, “A virtual hardware operating system for the Xilinx XC6200”, Proceedings of the 6th International Workshop on Field Programmable Logic (FPL´96), pp. 327-336, 1996.

[CACT08] http://www.hpl.hp.com/research/cacti/

[CaKP91] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching", Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 40-52, 1991.

[CGGR07] J.A. Clemente, C. González, J.L. García, J. Resano, and D. Mozos, "HW Implementation of a Task Manager for Reconfigurable Systems", Engineering of Reconfigurable Systems and Algorithms, pp. 71-77, 2007.

[CGRM08] J.A. Clemente, C. González, J. Resano, D. Mozos, "Task-graph management for reconfigurable multi-tasking systems", 4th International Workshop on Reconfigurable Communication Centric System-on-Chips (ReCoSoc), 2008.

[CoHa02] K. Compton, and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," ACM Comput. Surveys, Vol. 34, No. 2, pp. 171 – 210, 2002.

[CoSC08] S. Cosemans, W. Dehaene, F. Catthoor, "A 3.6pJ/Access 480MHz, 128Kbit on-Chip SRAM with 850MHz Boost Mode in 90nm CMOS with Tunable Sense Amplifiers to Cope with Variability", Solid-State Circuits Conference (ESSCIRC'08), pp. 278-281, 2008.

[CWGS98] S. Cadambi, J. Weener, S.C. Goldstein, H. Schmit, and D.E. Thomas, “Managing pipeline-reconfigurable FPGAs”, Proceedings of the ACM/SIGDA International Symposium on FPGAs, pp. 55-64, 1998.

[DeHo00] A. DeHon, “The Density Advantage of Configurable Computing”, IEEE Computer, Vol. 33, No. 4, 2000

[DiRW] R.P. Dick, D.L. Rhodes, W. Wolf "TGFF: task graphs for free", International Conference on Hardware Software Codesign, Proceedings of the 6th international workshop on Hardware/software codesign, pp. 97-101, 1998.

[DRLJ00] R.P. Dick, G. Lakshminarayana, A. Raghunathan, N.K. Jha, "Power analysis of embedded operating systems", Proceedings of the Design Automation Conference (DAC'00), pp. 312-315, 2000.

[DuYN97] J. Duato, S. Yalamanchili and L. Ni, "Interconnection Networks: An Engineering Approach", IEEE Computer Society Press, 1997.

[Elix05] http://www.electronicstalk.com/news/exi/exi000.html

[EsKO90] H. Eschenauer, J. Koski, and A. Osyczka. "Multicriteria Design Optimization", Springer-Verlag, 1990.

[FPCK97] R. Fromm, S. Perissakis, N. Cardwell, B. McGaughy, C. Kozyrakis, D. Patterson, T. Anderson, K. Yelick, "The energy efficiency of IRAM architectures", Proc. 24th Int. Symp. Computer Architecture, pp. 327-337, 1997.

[FuCo05] W. Fu, K. Compton, "An execution environment for reconfigurable computing", Proc. of Field-Programmable Custom Computing Machines (FCCM), pp.149-158, 2005.

[GaBh99] K.M. Gajjala Purna, and D. Bhatia, "Temporal partitioning and scheduling data flow graphs for reconfigurable computers," IEEE Transactions on Computers, Vol. 48, No. 6, pp. 579-590, 1999.

[GPHV09] V. Gheorghita, M. Palkovic, J. Hamers, A. Vandecappelle, S. Mamagkakis, T. Basten, L. Eeckhout, H. Corporaal, F. Catthoor, F. Vandeputte, K. De Bosschere, "System scenario based design of dynamic embedded systems", ACM Trans. on Design Automation for Embedded Systems (TODAES), Vol.14, 2009.

[GPHV09] S.V. Gheorghita, M. Palkovic, J. Hamers, A. Vandecappelle, S. Mamagkakis, T. Basten, L. Eeckhout, H. Corporaal, F. Catthoor, F. Vandeputte, K. De Bosschere, "System-scenario-based design of dynamic embedded systems", ACM Transactions on Design Automation of Electronic Systems (TODAES'09), Vol. 14, No. 1, 2009.

[Hart01] R. Hartenstein, "A decade of reconfigurable computing: A visionary retrospective", DATE'01, pp. 642-649, 2001.

[HCDV03] S.Himpe, F.Catthoor, G.Deconinck, J.Van Meerbergen, "MTG* and Grey-Box: modeling dynamic multimedia applications with concurrency and non-determinism", chapter in "System specification and design languages: best of FDL'02", Kluwer Academic Publishers, 2003.

[Ipfl05] www.ipflex.com

[JBVC05] M. Jayapala, F. Barat, T. Vander Aa, F. Catthoor, H. Corporaal, and G. Deconinck, "Clustered Loop Buffer Organization for Low Energy VLIW Embedded Processors", IEEE Transactions on Computers, Vol. 54, No. 6, pp. 672-683, 2005.

[JBVC05] M. Jayapala, F. Barat, T. Vander Aa, F. Catthoor, H. Corporaal, and G. Deconinck, "Clustered loop buffer organization for low energy VLIW embedded processors", IEEE Trans. Comput., Vol. 54, No. 6, pp. 672-683, 2005.

[JPEG] www.jpeg.org

[Kenn03] I. Kennedy, "Exploiting Redundancy to Speedup Reconfiguration of fan FPGA", FPL'033, pp. 262-271, 2003.

[KnSM99] J. Kneip, B. Schmale, and H. Moller, "Applying and Implementing the MPEG-4 Multimedia Standard", IEEE Micro, Vol. 19, Issue 6, pp. 66-74, 1999.

[KnSM99] J. Kneip, B. Schmale, and H. Möller, "Applying and implementing the MPEG-4 multimedia standard", IEEE Micro., Vol. 19, No. 6, pp. 66-74, 1999.

[LaLS99] M. Lajolo, M. Lazarescu, and A.L. Sangiovanni-Vincentelli, "A compilation-based software estimation scheme for hardware/software cosimulation", CODES, pp. 85-89, 1999.

[Leig92] F.T. Leighton. "Introduction to Parallel Algorithms and Architectures", Morgan Kaufmann Publishers, pp. 210-213, 1992.

[LeMi03] G. Lee, and G. Milne, “Building Run-times Reconfigurable Systems from Tiles”, Proceedings of the Field Programmable Logic Conference (FPL´03), pp. 252-261, 2003.

[LiCH00] Z. Li, K. Compton, S. Hauck, "Configuration Cache Management Techniques for FPGAs", IEEE FCCM'00, pp. 22-36, 2000.

[LiHa01] Z. Li, S. Hauck, "Configuration Compression for Virtex FPGAs", IEEE FCCM'01, pp. 147-159, 2001.

[LiHa02] Z. Li, S. Hauck, "Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation", FPGA'02, pp. 187-195, 2002.

[LPPA02] D.I. Lehn, K. Puttegowda, J.H. Park, P. Athanas, and M. Jones, "Evaluation of Rapid Context Switching on a CSRC Device", ERSA'02, pp. 209-215, 2002

[LPPA02] D. Lehn, K. Puttegowda, J. Park, P. Athanas, and M. Jones, "Evaluation of Rapid Context Switching on a CSRC Device", Proceedings of the International Conference on Engineering Reconfigurable Systems and Algorithms (ERSA 2002), pp. 209-215, 2002.

[LWMV02] R. Lauwereins, Ch. Wong, P. Marchal, J. Vounckx, P. David, S. Himpe, F. Catthoor, P. Yang, “Managing Dynamic Concurrent Tasks in Embedded Real-Time Multimedia Systems”, Proceedings of the International Symposium on System Synthesis (ISSS’02), pp. 112-119, 2002.

[M4if05] www.m4if.org

[MaCC01] J.M. Masgonty, C. Cserveny, C. Piguet, 'Low-power SRAM and ROM memories", Proc. Int. Workshop-Power and Timing Modeling, Optimization and Simulation, pp. 741-747, 2001.

[MaSC07] Z. Ma, D.P. Scarpazza, F. Catthoor, "Run-time Task Overlapping on Multiprocessor Platforms", Embedded Systems for Real-Time Multimedia (ESTIMedia 2007), pp. 47-52, 2007.

[MaTo09] S. Martello, P. Toth, "Knapsack Problems: Algorithms and Computer Implementations", J. Wiley & Sons, 1990.

[MBVL02] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins, "Interconnection Network enable Fine-Grain Dynamic Multi-Tasking on FPGAs", Proceedings of the Field Programmable Logic Conference (FPL), pp. 795-805, 2002.

[MeLJ98] P. Merino, J.C. López, and M. Jacome, "A Hardware Operating System for Dynamic Reconfiguration of FPGAs", Proceedings of the Field Programmable Logic Conference (FPL'98), pp. 431-435, 1998.

[Micr09] http://www.micron.com/products/dram/mobiledram/index

[MJSY02] P. Marchal, M. Jayapala, S.X. de Souza, P. Yang, F. Catthoor, G. Deconinck, "Matador: An Exploration Environment for System-Design", Journal of Circuits, Systems, and Computers, Vol. 11, No. 5, pp. 503-536, 2002.

[MKBS99] R. Maestre, F.J. Kurdahi, N. Bagherzadeh, H. Singh, R. Hermida, M. Fernandez, "Kernel Scheduling in Reconfigurable Computing", DATE'99, pp. 90-95, 1999.

[MMBM03] T. Marescaux, J-Y. Mignolet, A. Bartic, W. Moffat, D. Verkest, S. Vernalde, R. Lauwereins, “Networks on Chip as Hardware Components of an OS for

Reconfigurable Systems", Proceedings of the International Conference on Field Programmable Logic and Applications (FPL´03), pp. 595-605, 2003

[MMSY07] Z. Ma, P. Marchal, D.P. Scarpazza, P. Yang, C. Wong, J.I. Gómez, S. Himpe, C. Ykman-Couvreur, F. Catthoor, "Systematic Methodology for Real-Time Cost-Effective Mapping of Dynamic Concurrent Task-Based Systems on Heterogeneous Platforms", Springer, 2007.

[MNCV03] J.Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, R. Lauwereins, "Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip", Proceedings of the Design, Automation and Test in Europe Conference (DATE), pp.10986-10993, 2003.

[MPEG] www.mpeg.org

[Mura89] T. Murata, "Petri Nets: Properties, Analysis and Applications", Proceedings of the IEEE, Vol. 77, No 4, pp. 541-580, 1989.

[MWHD03] Z. Ma, C. Wong, S. Himpe, E. Delfosse, F. Catthoor, J. Vounckx, and G. Deconinck, "Task concurrency analysis and exploration of visual texture decoder on a heterogeneous platform", IEEE Workshop on Signal Processing Systems (SiPS 2003), pp. 245-250, 2003.

[Nara93] P. Narayanan, "Processor autonomy on SIMD architectures", International Conference on Supercomputing, pp. 127–136, 1993.

[NCVV03] V. Nollet, P. Coene, D. Verkest, S. Vernalde, R. Lauwereins, "Designing an Operating System for a Heterogeneous Reconfigurable SoC", Proceedings of the RAW'03 workshop, 2003.

[NMBV03] V. Nollet, J-Y. Mignolet, T.D. Bartic, D. Verkest, S. Vernalde, R. Lauwereins, "Hierarchical Run-Time Reconfiguration Managed by an Operating System for Reconfigurable Systems", Proceedings of the International Conference on Engineering Reconfigurable Systems and Algorithms (ERSA´03), pp. 81-87, 2003.

[NMVM04] V. Nollet, T. Marescaux, D. Verkest, J-Y. Mignolet, S. Vernalde, "Operating System controlled Network-on-Chip", Proceedings of the Design Automation Conference (DAC), pp. 256-259, 2004.

[NoBa04] J. Noguera, M. Badía, "Power-Performance Trade-Offs for Reconfigurable Computing", CODES+ISSS'04, pp. 116-121, 2004.

[NoBa04a] J. Noguera, M. Badía, "Multitasking on Reconfigurable Architectures: Microarchitecture Support and Dynamic Scheduling", ACM Transactions on Embedded Computing Systems, Vol. 3, No. 2, pp. 385-406, 2004.

[NoBa04b] J. Noguera, and R.M. Badía, ”Multitasking on Reconfigurable Architectures: Microarchitecture Support and Dynamic Scheduling”, ACM Transactions on Embedded Computing Systems, Vol. 3, No. 2, pp. 385-406, 2004.

[NoBa04b] J. Noguera, M. Badía, "Multitasking on reconfigurable architectures: microarchitecture support and dynamic scheduling", ACM Transactions on Embedded Computing Systems (TECS), Vol. 3, No. 2, pp. 385-406, 2004.

[PaMW04] J.H. Pan, T. Mitra, W.F. Wong, "Configuration Bitstream Compression for Dynamically Reconfigurable FPGAs", ICCAD'04, pp. 766- 773, 2004.

[PTCC09] A. Portero, G. Talavera, J. Carrabina, and F. Catthoor, "Data-Dominant Application implementation in Multiplatform for energy-flexibility space Exploration", IEEE Transactions on Very Large Scale Integration Systems, 2009 (to be published).

[PTMM09] A. Portero, G. Talavera, M. Moreno, B. Martinez, J. Saiz, M. Monton, J. Carrabina, and F. Catthoor, "Multiplatform Video Encoder implementation for energy-flexibility space Exploration", Embedded Hardware Design (Microprocessors and Microsystems), 2009 (to be published).

[QuSN06] Y. Qu, J. Soininen, J. Nurmi, "A Parallel Configuration Model for Reducing the Run-Time Reconfiguration Overhead", DATE'06, pp. 965- 969, 2006.

[RCGG07] J. Resano, J.A. Clemente, C. Gonzalez, J.L. Garcia, D. Mozos, "HW implementation of an execution manager for reconfigurable systems", ERSA'07, pp. 71-77, 2007.

[RiSB07] F. Rivera, M. Sanchez-Elez, and N. Bagherzadeh, "Configuration and data scheduling for executing dynamic applications onto multi-context reconfigurable architectures", Engineering of Reconfigurable Systems and Algorithms (ERSA), pp. 85–91, 2007.

[RJLA09] P. Raghavan, M. Jayapala, A. Lambrechts, J. Absar, and F. Catthoor, "Playing the trade-off game: Architecture exploration using COFFEE", ACM transactions on design automation of electronic systems, Vol. 14, No. 2, Article 36, 2009 (to be published).

[RLGC98] C. R. Rupp, M. Landguth, T. Garverick, E. Comersall, H. Holt, J.M. Arnold, and M. Gokhale, “The NAPA adaptive processing architecture”, IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 28-37, 1998.

[RLMC06] P. Raghavan, A. Lambrechts, M. Jayapala, F. Catthoor, and D. Verkest, "Distributed loop controller architecture for multi-threading in uni-threaded VLIW processors", Design, Automation, and Test in Europe, pp. 339-344, 2006.

[RMCC08] J. Resano, D. Mozos, F. Catthoor, J.A. Clemente, C. Gonzalez, "Efficiently scheduling run-time reconfigurations", Transactions on Design Automation of Electronic Systems, Vol. 13, No. 4, pp. 58.1-58.12, 2008.

[RMMS08] S. Román, H. Mecha, D. Mozos, and J. Septién, "Constant Complexity Scheduling for Hardware Multitasking in Two Dimensional Reconfigurable Field-Programmable Gate Arrays", IET Computer Digital Techniques, Vol. 2, pp. 401-412, 2008.

[RMVC05] J. Resano, D. Mozos, D. Verkest, and F. Catthoor, "A reconfiguration manager for dynamically reconfigurable hardware", IEEE Des. Test, Vol. 22, No. 5, pp. 452-460, 2005.

[RSHB08] F. Rivera, M. Sánchez-Élez, R. Hermida, and N. Bagherzadeh, "Scheduling Methodology for Conditional Execution of Kernels onto Multi-Context Reconfigurable Architectures", IET Computers & Digital Techniques, Vol. 2, No 3, pp. 199-213, 2008.

[RVMC04] J. Resano, D. Verkest, D. Mozos, F. Catthoor, S. Vernalde, "Specific scheduling support to minimize the reconfiguration overhead of dynamically

Reconfigurable Hardware", IEEE 41 Design Automation Conference, pp. 119-124, 2004.

[RVMV04] J. Resano, D. Verkest, D. Mozos, S. Vernalde, F. Catthoor, "A hybrid design-time/run-time scheduling flow to minimise the reconfiguration overhead of FPGAs", Elsevier J. Microproc. Microsyst., Vol. 28, No. 5-6, pp. 291-301, 2004.

[Scar06] D.P. Scarpazza, "A Source-level Estimation and Optimization Methodology for Execution Time and Energy Consumption of Embedded Software", PhD thesis, 2006.

[ScBr06] D.P. Scarpazza, and C. Brandolese, "A fast, dynamic, Fine-detail, source level technique to estimate the energy consumed by embedded software on single issue processor cores" Journal of Low-Power Electronics, Vol. 2, No. 2, pp. 129-139, 2006.

[SDLB03] M. Sanchez-Elez, H. Du, N. Tabrizi, Y. Long, N. Bagherzadeh, M. Fernandez, "Algorithm Optimizations and Mapping Scheme for Interactive Ray Tracing on a Reconfigurable Architecture", Computer & Graphics, Vol. 27, No 1, pp. 701-713, 2003.

[SFHB05] M. Sánchez-Élez, M. Fernández, R. Hermida, and N. Bagherzadeh, "A Low Energy Data Management for Multi-Context Reconfigurable Architectures", Springer, pp. 145-155, 2005.

[SFHB05] M. Sánchez-Élez and M. Fernández and R. Hermida and N. Bagherzadeh, "A Low Energy Data Management for Multi-Context Reconfigurable

Architectures", New Algorithms, Architectures and Applications for Reconfigurable Computing, Springer, 145-155, 2005.

[SFHM02] M. Sanchez-Elez, M. Fernández, R. Hermida, R. Maestre, N. Bagherzadeh, F. Kurdahi, "A Complete Data Scheduler for Multi-Context Reconfigurable Architectures", DATE Proceedings, 2002.

[ShJh02] L. Shang, N.K. Jha, "Hw/Sw Co-synthesis of Low Power Real-Time Distributed Embedded Systems with Dynamically Reconfigurable FPGAs", ASP-DAC'02, pp. 345-360, 2002.

[ShJh02] L. Shang, and N.K. Jha, "Hw/Sw co-synthesis of low power real-time distributed embedded systems with dynamically reconfigurable FPGAs" Asia and South Pacific Design Automation Conf., (ASP-DAC'02), pp. 345-360, 2002.

[Sing00] H. Singh, "Reconfigurable Architectures for Multimedia and Data-Parallel Application Domains", PhD thesis, EECS Department, University of California, Irvine, 2000.

[SLLK00] H. Singh, M. Lee, G. Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M.Ch. Filho, “MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications”, IEEE Transactions on Computers, Vol. 49, No. 5, pp. 465-481, 2000.

[SLLK00] H. Singh, M.H. Lee, G. Lu, F.J. Kurdahi, N. Bagherzadeh, "MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications", IEEE Transactions on Computers, Vol. 49, No. 5, pp. 465-481, 2000.

[Sony05] www.sony.net/Products/SC-HP/cx_news/vol42/pdf/sideview42.pdf

[SRNC06] D.P. Scarpazza, P. Raghavan, D. Novo, F. Catthoor, and D. Verkest, "Software simultaneous multi-threading, a technique to exploit task-level parallelism to improve instruction- and data-level parallelism", PATMOS, pp. 12-23, 2006.

[StRa91] J. Stankovic, and K. Ramamritham, "The spring kernel: A new paradigm for real-time systems", IEEE Software, Vol. 8, No. 3, pp. 62-72, 1991.

[StWP04] C. Steiger, H. Walder, and M. Plazner, “Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks”, IEEE Transactions on Computers, Vol. 53, No. 11, pp. 1393-1407, 2004.

[SuNG01] S. Sudhir, S. Nath, and S.C. Goldstein, "Configuration Caching and Swapping", Proc. of FPL'01, pp. 192-202, 2001.

[SVKS01] P. Schaumont, I. Verbauwhede, K. Keutzer, and M. Sarrafzadeh “A quick safari through the reconfiguration jungle”, Proceedings of the 38th conference on Design automation, pp. 172-177, 2001.

[TCJW97] S. Trimberger, D. Carberry, A. Johnson, J. Wong, "A time multiplexed FPGA", FCCM'97, pp.22-29, 1997.

[TCJW97] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. “A time multiplexed FPGA”, Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '97), pp. 22-29, 1997.

[TeIn09] http://focus.ti.com/general/docs/dsnsuprt.tsp

[ThCa00] F. Thoen, F. Catthoor, "Modeling, Verification, and Exploration of Task-Level Concurrency of Real-Time Embedded Systems", Kluwer Academic Publishers, 2000.

[VaSo07] S. Vassiliadis, D. Soudris, “Fine and Coarse Grain Reconfigurable Computing”, Springer, 2007.

[Vira09] http://www.viragelogic.com/render/content.asp?id=292

[ViVa06] K.N. Vikram, V. Vasudevan, "Mapping Data-Parallel Tasks Onto Partially Reconfigurable Hybrid Processor Architectures", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 14, No. 9, pp.1010-1023, 2006.

[VJBD04] T. Vander Aa, M. Jayapala, F. Barat, G. Deconinck, R. Lauwereins, F. Catthoor, and H. Corporaal, "Instruction Buffering Exploration for Low Energy VLIWs with Instruction Clusters", Asian Pacific Design and Automation Conference 2004 (ASP-DAC’2004), pp. 825-830, 2004.

[VoMa06] N.S. Voros, and K. Masselos, "OCAPI-XL Based Approach", chapter 6 in System Level Design of Reconfigurable Systems-on-Chip, Springer US, pp. 133-151, 2006.

[WMYP01] Ch. Wong, P. Marchal, P. Yang, A. Prayati, F. Catthoor, R. Lauwereins, D. Verkest, H. De Man, "Task concurrency management methodology to schedule the MPEG4 IM1 player on a highly parallel processor platform", Proceedings of the International Conference on Hardware/Software Codesign (CODES'01), pp. 170-175, 2001.

[XDSP09] http://www.xilinx.com/tools/sysgen.htm

[XEDK09] http://www.xilinx.com/tools/platform.htm

[XEDK09] http://www.xilinx.com/tools/platform.htm

[Xili00] Xilinx Inc., "Configuration Architecture of Virtex Field Programmable Arrays Product Description", 2000.

[Xili05] Xilinx Inc., "Virtex-II Pro and Virtex-II Pro X FPGA User Guide", 2005.

[Xili09] http://www.xilinx.com/products/devices.htm

[Xili09] www.xilinx.com

[XilV-5] http://www.xilinx.com/publications/xcellonline/xcell_59/index.htm

[XiMB09] http://www.xilinx.com/support/documentation/sw_manuals/edk92i_mb_ref_guide.pdf

[XISE09] http://www.xilinx.com/tools/logic.htm

[XiSG09] http://www.xilinx.com/tools/sysgen.htm

[Xpower] http://www.xilinx.com/products/design_tools/logic_design/verification/xpower.htm

[XVirII] http://www.xilinx.com/onlinestore/v2_boards.htm

[YaCa03] P. Yang, F. Catthoor, "Pareto-optimization-based run-time task scheduling for embedded systems", Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'03), pp. 120-125, 2003.

[YaCa04] P. Yang, F. Catthoor, "Dynamic Mapping and Ordering Tasks of Embedded Real-Time Systems on Multiprocessor Platforms", Proceedings of the

Software and Compilers for Embedded Systems Conference (SCOPES 2004), pp. 167-181, 2004.

[YaGe93] T. Yang, A. Gerasoulis, "List scheduling with and without communication delays", Parallel Computing, Vol. 19, No. 12, pp. 1321-1344, 1993.

[YDCV00] P. Yang, D. Desmet, F. Catthoor, D. Verkest, "Dynamic scheduling of concurrent tasks with cost performance trade-off", Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'00), pp. 103-109, 2000.

[YLWM02] P. Yang, R. Lauwereins, C. Wong, P. Marchal, J. Vounckx, P. David, S. Himpe, and F. Catthoor, "Managing dynamic concurrent tasks in embedded real-time multimedia systems", In Proceedings of ISSS 15th International Symposium on System Synthesis, pp. 112-119, 2002.

[YMHB00] Z.A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, “Chimaera: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Function Unit”, Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00), pp. 225-235, 2000.

[YWMC01] P. Yang, Ch. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest, and R. Lauwereins, "Energy-Aware Runtime Scheduling for Embedded-Multiprocessors SOCs", IEEE Journal on Design&Test of Computers, pp. 46-58, 2001.

Yo... he visto cosas que vosotros no creeríais... atacar naves en llamas más allá de Orión, he visto rayos C brillar en la oscuridad cerca de la puerta Tannhäuser.

Todos esos momentos se perderán en el tiempo como lágrimas en la lluvia.

Es hora de morir.

— Roy Batty (Blade Runer, 1982)